UNIVERSITE BRETAGNE LOIRE

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Brice NÉDELEC

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'Université de Nantes
sous le sceau de l'Université Bretagne Loire

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique et applications
Unité de recherche : Laboratoire d'informatique de Nantes-Atlantique (LINA)

Soutenue le 5 Octobre 2016

Édition collaborative décentralisée dans les navigateurs

JURY

Président :	**M. Marc Gelgon**, Professeur, Polytech Nantes
Rapporteurs :	**M^me^ Anne-Marie Kermarrec**, Directrice de recherches INRIA, INRIA Rennes **M. Peter Van Roy**, Professeur, Université catholique de Louvain
Examinateurs :	**M. Gérald Oster**, Maître de conférences, Université de Lorraine **M. Marc Shapiro**, Directeur de recherches INRIA/LIP6, LIP6 Paris
Directeur de thèse :	**M. Pascal Molli**, Professeur, Université de Nantes
Co-directeur de thèse :	**M. Achour Mostéfaoui**, Professeur, Université de Nantes

Remerciements

Je remercie Pascal Molli et Achour Mostéfaoui – mes « Grands Maîtres » – de m'avoir accueilli dans leur équipe ; de m'avoir guidé et conseillé durant ces quatre années qui furent riches en enseignements. Je remercie chacun des membres de l'équipe pour leur amicale présence. En particulier, Emmanuel Desmontils qui fut pendant quelques temps mon encadrant, et qui, bien que s'étant retiré de cette charge, n'en demeura pas moins généreux en conseils et amitiés.

Je remercie mes compagnons de misère, Adrien, Gabriela et Pauline pour leur soutien indéfectible, les nombreux cafés et repas partagés ; Julian for the few months of shared work that introduced me to a new field of research. Plus éloignée mais non moins importante : la cohorte de l'école des Mines comprenant Alexandre, Florent, Jonathan, Ronan et Simon ; avec lesquels je partage une passion pour la musique et les quarts de cercle.

Enfin, merci à mes amis, Adrien, Alexandre, Amélie, Clémentine et Maxime pour ne citer qu'eux ; et aux membres de ma famille qui m'ont accompagné durant ces longues années.

Ces travaux ont été financés par le projet ANR ConcoRDanT (`ANR-10-BLAN-0208`), le projet ANR SocioPlug (`ANR-13-INFR-0003`) et le projet DeSceNt accordé par le programme « Laboratoires d'Excellence » du Labex CominLabs (`ANR-10-LABX-07-01`).

Une partie des expériences présentées dans ce manuscrit furent réalisées sur le banc d'essai Grid'5000, maintenue par la communauté scientifique gérée par l'Inria et incluant le CNRS, RENATER, ainsi que de nombreuses universités et organismes (voir `www.grid5000.fr`).

Table des matières

Table des figures

Liste des algorithmes

Liste des tableaux

1

Introduction

Sommaire

1.1 Contexte

L'édition collaborative concerne toutes les activités effectuées en groupe dans le but de produire un document [25, 51]. Grâce à un effort collectif, les rédacteurs bénéficient de multiples points de vue et deviennent plus impliqués dans l'écriture [79]. Les documents en résultant sont de meilleure qualité. Ainsi l'encyclopédie Wikipédia compte des millions d'articles rédigés par des millions d'auteurs et sa version anglaise possède une fiabilité comparable à celle de l'Encyclopædia Britannica [39].

Les éditeurs collaboratifs de texte sont simplement des outils facilitant l'écriture de documents par plusieurs auteurs. Ils présentent un texte dont le contenu est modifiable par un groupe de collaborateurs. Chaque modification est transmise à tous les collaborateurs qui peuvent alors les voir et écrire en conséquence. Grâce à ces outils, la tâche d'écriture

peut être répartie selon le temps et l'espace [24, 43, 51]. Ainsi, il n'est plus impératif de se retrouver dans la même pièce ni même de se retrouver au même moment pour participer à la rédaction du document.

Si le premier éditeur collaboratif date de 1968 [27], leur adoption massive par le public n'est que récente et émane principalement des éditeurs Web temps réel [68, 83] tels que Google Docs [40] ou ShareLaTeX [81]. Grâce au Web, il est possible de créer et d'éditer facilement un document depuis un ordinateur de bureau, un téléphone portable ou une tablette tactile. Un simple lien permet de le partager avec des amis ou des collègues. L'édition collaborative devient si aisée qu'elle est à la portée de tous. Cependant, de nos jours, un petit groupe d'entreprises détient une large portion du Web. Les applications Web populaires tels que les éditeurs collaboratifs sont maintenues par quelques grandes sociétés. Cette organisation éminemment centralisée, où quelques serveurs sont en charge d'un nombre titanesque de clients, pose des problèmes d'ordre éthique sur la confidentialité, la propriété et la censure [19, 37, 82] ; et des problèmes d'ordre technique sur le passage à l'échelle et la résilience aux pannes.

Pour ces raisons, la redécentralisation du Web, et plus généralement de l'internet, a reçu beaucoup d'attention ces derniers temps [11, 14, 63, 107]. La décentralisation constituerait un progrès vers une solution aux problèmes liés à l'éthique. Par exemple, la propriété des documents pourrait être rendue à ceux qui les rédigent. La décentralisation constituerait une solution partielle aux problèmes d'ordre technique. Par exemple, elle permettrait d'alléger l'infrastructure des services Web. Chaque client devenant également un serveur, le poids de l'édition s'en trouverait réparti parmi les participants au lieu du Nuage [66].

La facilité d'accès apportée par le Web au service de la décentralisation inspire également de nouveaux usages de plus grande ampleur et plus dynamiques. Par exemple, un document n'est plus cantonné aux petits groupes : le Web entier peut participer à son élaboration, donner ses avis, participer à sa rédaction. Lors d'un cours de formation en ligne (*MOOC*) [15], la prise de notes peut rassembler des milliers de personnes à son commencement. Ensuite, nombre d'étudiants quittent le cours par manque d'intérêt. Enfin, beaucoup d'étudiants réintègrent le groupe d'édition lorsque l'examen final approche. Par conséquent, les éditeurs collaboratifs doivent supporter des groupes dont les dimensions ont largement augmenté, et dont le nombre de collaborateurs fluctue à chaque instant.

Se pose donc la question de la faisabilité d'une telle application : **L'édition collaborative temps réel est-elle possible sur le Web, sans l'intervention d'un tiers et sans limites quant aux dimensions du système ?**

1.2 Questions de recherche

L'édition collaborative répartit l'écriture selon deux dimensions : le temps et l'espace [24, 43, 51]. La figure 1.1 résume cette répartition. Dans ce manuscrit, nous nous intéressons

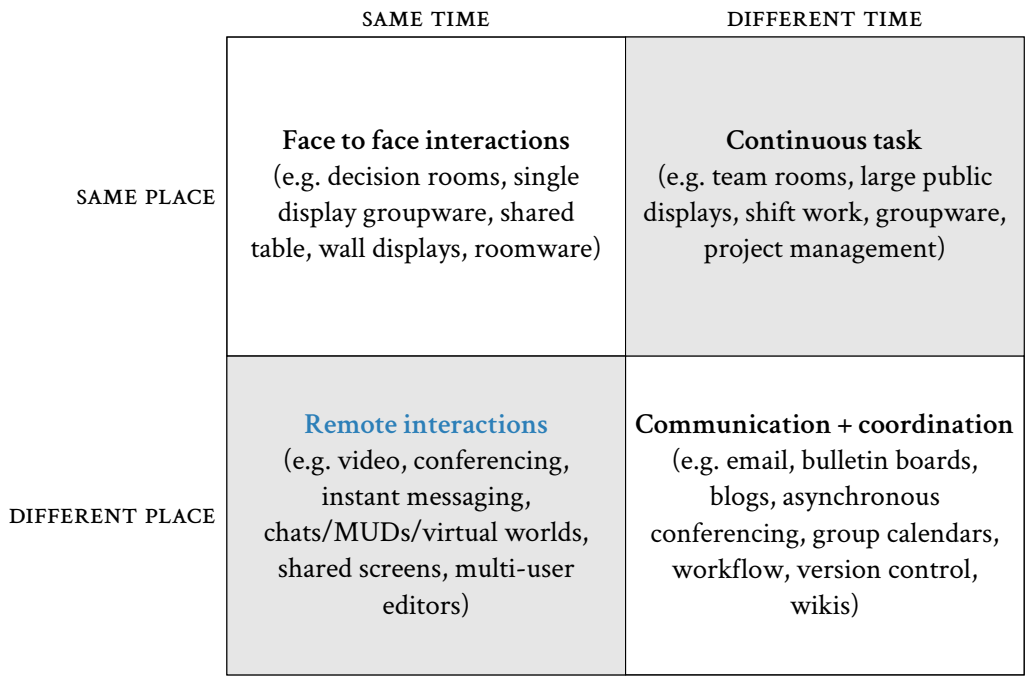


FIGURE 1.1 – Matrice de répartition des systèmes collaboratifs [51].

particulièrement aux collaborateurs connectés en même temps en des lieux différents correspondant à la problématique du temps réel. Dans ce contexte, « temps réel » signifie que les modifications doivent être perçues au plus tôt [25]. La tolérance d'un utilisateur varie selon les outils collaboratifs à sa disposition. Une modification locale doit être observée immédiatement. En revanche, une modification distante est plus lente à être observée.

Un éditeur de texte tel que Microsoft Word [67] ou Emacs [92] permet à un utilisateur de créer et d'éditer un document. Un éditeur de texte collaboratif temps réel [26] étend la rédaction d'un document à un groupe d'utilisateurs potentiellement distants les uns des autres. Par exemple, la figure 1.2 présente une session d'édition comprenant 6 instances d'un même éditeur collaboratif réparties à travers le monde éditant un même document. Chacun possède son éditeur, voit les modifications effectuées par ses collaborateurs sur le document, effectue ses propres changements en insérant ou en supprimant du contenu. La mise en place de ces fonctionnalités temps réel requiert (i) un moyen de représenter les documents en mémoire de telle sorte que tous les éditeurs affichent un même document lorsqu'ils ont reçu les mêmes modifications [16, 90] ; (ii) un moyen de communication fiable entre les éditeurs fonctionnant sur des machines potentiellement distantes.

Afin d'augmenter la disponibilité du document et de diminuer le temps de réponse lors d'une modification, les éditeurs collaboratifs actuels suivent le principe de la réplication op-

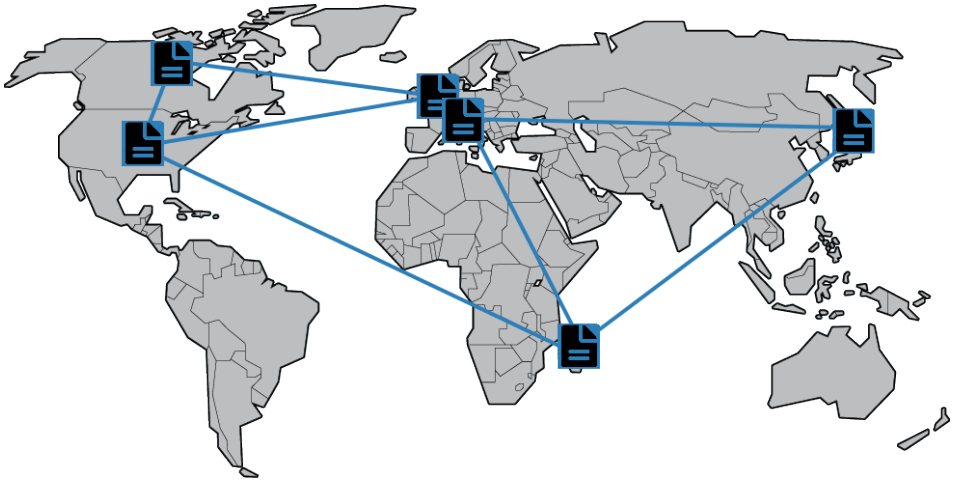


FIGURE 1.2 – Édition collaborative décentralisée répartie à travers le monde.

timiste [88]. Chaque éditeur collaboratif possède une copie locale d'un document – pouvant être représenté simplement par une séquence de caractères autorisant les deux opérations suivantes : (a) l'insertion d'un caractère à une position dans la séquence et (b) la suppression du caractère à une position dans la séquence. Lorsque l'utilisateur effectue une opération (i) elle est directement répercutée sur sa copie avant (ii) d'être envoyée au reste des éditeurs où (iii) elle est intégrée. Les séquences répliquées convergent vers un état identique sous l'hypothèse d'une diffusion fiable des opérations, i.e., tous les éditeurs reçoivent toutes les modifications.

Pour assurer la convergence des répliques, l'utilisation des structures de données « classiques » n'est pas directement possible. Par exemple, la figure 1.3 présente une situation où un auteur insère un caractère sur la réplique 1 en début de document pendant qu'un autre supprime un caractère sur la réplique 3 au même endroit. Le premier auteur voit le caractère qu'il vient d'insérer supprimé. Quant à l'état final de la réplique 2, il dépend de l'ordre de réception des opérations. Ici, la suppression arrivant avant l'insertion, la séquence répliquée devient `QERTY`. Afin d'éviter de telles divergences dans l'état des répliques, de nouvelles structures et de nouveaux algorithmes adaptés au contexte réparti doivent être employés.

Une première famille d'approches consiste à transformer les arguments de toute opération afin qu'elle considère les opérations intégrées concurremment [94]. Dans l'exemple précédent, lors de l'intégration de la suppression par le premier éditeur, celui-ci doit réaliser qu'un caractère a été ajouté en tête concurremment. Il doit donc modifier les arguments de l'opération reçue afin que le caractère `W` soit supprimé, et non plus le caractère `Q`. Toutes les répliques convergent alors vers la séquence `QERTY`. Détecter ces cas concurrents nécessite de communiquer, avec chaque opération, des données dont la croissance

FIGURE 1.3 – Répliques divergentes d'un document.

est linéaire [18, 96]. Cette contrainte confine l'emploi de ces approches aux petits groupes d'utilisateurs.

Une seconde famille d'approches consiste à utiliser des structures de données dont les opérations sont commutatives [90] et ne souffrent donc pas de la concurrence. Pour que les opérations commutent, ces approches associent à chaque caractère de la séquence un identifiant unique et immuable. Nous distinguons deux types de structures se différenciant par la nature des identifiants considérés :

Les identifiants cachés [80]. L'opération de suppression de caractères se contente de cacher le caractère ciblé par l'utilisateur. La mémoire consommée par une réplique croît de manière monotone par rapport au nombre d'insertions. La structure répliquée d'un document vide peut contenir des milliers d'identifiants cachés correspondant à des caractères supprimés. Afin d'y remédier, un protocole de ramasse-miettes réparti [1] doit être exécuté régulièrement pour purger la structure des identifiants cachés. Malheureusement, cela s'avère extrêmement coûteux [1].

Les identifiants de taille variable [105]. Les identifiants sont des listes dont la taille est fixée à la génération. Cependant, la fonction de génération d'identifiants alloue des identifiants dont la taille peut dépendre linéairement du nombre d'insertions dans la séquence [105]. Puisque chaque identifiant doit être envoyé à toutes les autres répliques, le trafic généré peut augmenter rapidement. Afin d'y remédier, un protocole de relocalisation des identifiants doit être exécuté régulièrement [112]. Ces protocoles reviennent à obtenir un consensus dans un contexte réparti. Malheureusement, cela s'avère extrêmement coûteux [72].

Cela pose la première question de recherche : **Afin d'éviter tout protocole additionnel de relocalisation des identifiants, comment allouer ces identifiants de sorte que leur taille soit directement sous-linéaire ?**

Afin d'assurer la convergence de toutes les répliques du document vers un état équivalent, les éditeurs collaboratifs font l'hypothèse d'une communication fiable : chacune des modifications doit être reçue par l'ensemble des éditeurs collaboratifs. Une manière effi-

cace de mettre en place cette diffusion fiable et décentralisée consiste à suivre le principe de dissémination épidémique[1] où (i) l'éditeur effectuant un changement choisit un sous-ensemble d'éditeurs possédant une réplique du document et leur envoie la modification ; (ii) lorsqu'un éditeur reçoit un tel message, il choisit à son tour un sous-ensemble d'éditeurs auxquels faire suivre le message. Les changements atteignent tous les éditeurs par transitivité. La figure 1.2 montre que chaque éditeur collaboratif est connecté à d'autres éditeurs – ses voisins – possédant une réplique du document. Ici, l'éditeur localisé aux États-Unis communique avec les éditeurs canadien, français et malgache. Une opération émise par le premier peut transiter par l'éditeur localisé à Madagascar avant de parvenir à l'éditeur japonais.

Notre contexte nous expose à deux contraintes :

(i) L'éditeur collaboratif doit fonctionner dans les navigateurs Web sans installations d'aucune sorte (e.g. extensions). Depuis peu, les navigateurs sont capables de communiquer entre eux après avoir suivi d'une procédure de connexion complexe [45]. L'éditeur doit donc supporter ce mode de connexion.

(ii) L'éditeur collaboratif doit maintenir un voisinage dont la taille doit permettre de gérer aussi bien les petits groupes que les grands. La taille de voisinage est déterminante puisque lorsqu'elle augmente, le trafic généré augmente au risque de dépasser les capacités d'émission des éditeurs ; lorsqu'elle diminue, le risque que les opérations ne parviennent plus à tous les éditeurs augmente. L'éditeur doit supporter les sessions d'édition où le nombre de participants fluctue fortement au cours du temps.

Les protocoles d'échantillonnage aléatoire de pairs [50] permettent à chaque éditeur de peupler son voisinage d'éditeurs choisis aléatoirement parmi les éditeurs impliqués dans la rédaction du document. Entre autres, ces protocoles assurent une forte résilience aux arrivées et aux départs dans la session collaborative – comportement fréquent dans le contexte Web. Cependant, la grande majorité de ces protocoles fournissent des voisinages configurés *a priori*, et donc, de taille constante [29, 50, 59, 97, 99]. En d'autres termes, le développeur doit connaître le nombre de participants impliqués dans la rédaction d'un document afin de configurer idéalement la taille de voisinage [28]. Pour ne pas renoncer aux grands groupes, le voisinage doit être surdimensionné. Il en résulte un trafic plus élevé. Le seul protocole permettant d'adapter automatiquement la taille du voisinage à la taille de la session d'édition se nomme Scamp [36]. Toutefois, ce dernier est inadapté à la procédure complexe d'établissement de connexion disponible dans les navigateurs Web.

Cela pose la seconde question de recherche : **Comment adapter efficacement le voisinage de chaque éditeur collaboratif Web au nombre effectif de collaborateurs ?**

[1] ou propagation de rumeurs.

1.3 Contributions

La première question de recherche nous place dans le contexte de la réplication optimiste [23, 88], i.e., un schéma de réplication où le document partagé est copié chez chaque utilisateur afin de fournir réactivité et disponibilité. La structure de séquence constitue une abstraction proche du document. Par conséquent, nous nous intéressons particulièrement à un type de données pour séquences dont les opérations commutent par nature [16, 89, 90, 111]. Ce type présente l'avantage de supporter efficacement les opérations concurrentes [3, 4]. Ces structures de données associent un identifiant unique et immuable à chaque élément de la séquence. Nous proposons LSeq [75, 76], une fonction d'allocation d'identifiants. Les identifiants générés ont une taille bornée polylogarithmiquement par rapport au nombre d'insertions effectuées dans la séquence. Nous définissons les conditions sous lesquelles s'applique cette borne et en fournissons la preuve. Au travers de simulations, nous validons LSeq et sa complexité.

Afin de répondre à la seconde question de recherche, nous nous sommes interessés aux protocoles d'appartenance aux réseaux (*membership*). Plus particulièrement à ceux dont la topologie résultante possède des propriétés similaires à celles des graphes aléatoires [28]. Parmi ces propriétés se trouve la capacité à gérer les connexions et déconnexions fréquentes. Ces protocoles sont regroupés sous l'appellation de « protocoles d'échantillonnage aléatoire de pairs » [47, 50] et servent de base à de nombreux protocoles répartis [22, 30, 69]. Nous proposons une nouvelle approche nommée Spray [77] appartenant à cette famille. Comparé à l'état de l'art [29, 35, 50, 59, 97, 99], Spray s'adapte automatiquement à la taille du réseau en suivant une progression logarithmique par rapport à la taille globale du réseau tout en supportant les contraintes imposées par la procédure complexe d'établissement de connexion dans les navigateurs Web. Nous mettons cela en évidence grâce à des simulations. Les protocoles de diffusion épidémique [13] de messages (*gossip*) qui en dépendent bénéficient à leur tour de cette capacité à s'adapter. Ainsi, le trafic généré peut s'adapter à la taille réelle du réseau.

Afin de répondre à la problématique générale de ce manuscrit, nous avons réuni les deux approches LSeq et Spray dans un éditeur collaboratif décentralisé nommé Crate [74]. Ce dernier fonctionne directement dans les navigateurs Web. Pour peu que l'utilisateur ait un accès à l'internet, Crate lui permet l'édition de documents n'importe quand, n'importe où, avec autant de collaborateurs qu'il le souhaite, sans fournisseur tiers. Comparé à l'état de l'art [32, 40, 53, 58, 78], (i) Crate constitue un premier pas vers une solution aux problèmes liés à la confidentialité ; (ii) Crate résout les problèmes de passage à l'échelle ; (iii) Crate conserve la facilité d'utilisation propre aux éditeurs Web. Nous validons Crate aux travers d'expériences réunissant jusqu'à 600 éditeurs sur le banc d'essai Grid'5000. Les résultats confirment le facteur logarithmique de Spray et la taille polylogarithmique des identifiants générés par LSeq.

1.4 Plan du manuscrit

Le reste du manuscrit est découpé en 4 chapitres :

Le chapitre 2 décrit la structure de données répliquée représentant le document partagé. Il présente les approches appartenant à la réplication dite optimiste [23, 88] : les approches à transformées opérationnelles [95, 96] et les approches à structures de données sans conflits [16, 90]. Le chapitre définit le problème scientifique à résoudre. Ensuite, il détaille LSeq [75, 76], une fonction d'allocation d'identifiants conçue pour le second type d'approches et dont la taille des identifiants est polylogarithmique par rapport au nombre d'insertions effectuées dans le document. Le reste du chapitre s'attache à valider cette propriété.

Le chapitre 3 commence par décrire les protocoles qui nous permettent de former les sessions d'édition : les protocoles d'échantillonnage aléatoire de pairs [47, 50]. Ceux-ci fournissent une vue partielle du réseau à chaque membre. Cette vue peut être utilisée afin de communiquer avec l'ensemble du réseau. L'état de l'art se décompose en deux familles : (i) Les vues partielles sont de taille constante configurée *a priori* [29, 50, 59, 97, 99] ; (ii) Les vues partielles s'adaptent à la volée aux dimensions du réseau [35, 36]. Le chapitre définit le problème scientifique à résoudre. Ensuite, il décrit le fonctionnement de Spray [77] appartenant à la seconde famille d'approches. Le cycle de vie d'un pair est décomposé entre le moment où il rejoint, le temps où il est membre, et le moment où il se retire. Tout au long de ce cycle, le système dans sa globalité doit rester cohérent sur la taille des vues partielles ainsi que sur leur contenu. Le chapitre décrit les propriétés assurées par Spray. Le chapitre montre également les avantages obtenus par un système de diffusion de messages profitant de Spray.

Le chapitre 4 décrit Crate [74], un éditeur collaboratif temps réel décentralisé fonctionnant dans les navigateurs Web. Tout d'abord, le chapitre décrit les composants employés par Crate. Profitant des propriétés de Spray et de LSeq, Crate parvient à adapter le trafic généré par l'édition à la taille de la session et à la taille du document. Ensuite, des expérimentations à large échelle confirment cette analyse.

Le chapitre 5 clôt ce manuscrit et décrit quelques perspectives ouvertes par ce type d'application ainsi que les défis qu'il reste à relever.

1.5 Publications

Crate : Writing Stories Together in our Browsers
— Brice Nédelec, Pascal Molli, and Achour Mostefaoui

Demo paper – Proceedings of the 25th International Conference Companion on World Wide Web (ACM WWW'16), 2016

Abstract : Real-time collaborative editors are common tools to distribute work across space, time, and organizations. Unfortunately, mainstream editors such as Google Docs rely on central servers and raise privacy and scalability issues. Crate is a real-time decentralized collaborative editor that runs directly in web browsers thanks to WebRTC. Compared to the state-of-the-art, Crate is the first real-time editor that only requires browsers in order to support collaborative editing and to transparently handle from small to large groups of users. Consequently, Crate can also be used in massive online lectures, TV shows or large conferences to allow users to share their notes. Crate's properties rely on two scientific results : (i) a replicated sequence structure with sub-linear upper bound on space complexity ; this prevents the editor from running costly distributed garbage collectors, (ii) an adaptive peer sampling protocol ; this prevents the editor from oversizing routing tables, hence from letting small networks pay the price of large networks. This paper describes Crate, its properties and its usage.

Spray : an Adaptive Random Peer Sampling Protocol
— Brice Nédelec, Julian Tanke, Davide Frey, Pascal Molli, and Achour Mostefaoui

Technical report, 2015

Abstract : The introduction of WebRTC has opened a new playground for large-scale distributed applications consisting of large numbers of communicating web browsers. In this context, gossip-based peer sampling protocols appear as a particularly promising tool thanks to their inherent ability to build overlay networks that can cope with network dynamics. However, the dynamic nature of browser-to-browser communication combined with the connection establishment procedures that characterize WebRTC make current peer sampling solutions inefficient or simply unreliable. In this paper, we address the limitations of current peer sampling approaches by introducing Spray, a novel peer-sampling protocol designed to avoid the constraints introduced by WebRTC. Unlike most recent peer sampling approaches, Spray has the ability to adapt its operation to networks that can grow or shrink very rapidly. Moreover, by using only neighbor-to-neighbor interactions, it limits the impact of the threeway connection establishment process that characterizes WebRTC. Our experiments demonstrate the ability of Spray to adapt to dynamic networks and highlight its efficiency improvements with respect to existing protocols.

LSeq : an Adaptive Structure for Sequences in Distributed Collaborative Editing
– Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils

Research paper – Best student paper award – Proceedings of the 2013 ACM Symposium on Document Engineering (ACM DocEng'13), 2013

Abstract : Distributed collaborative editing systems allow users to work distributed in time, space and across organizations. Trending distributed collaborative editors such as Google Docs, Etherpad or Git have grown in popularity over the years. A new kind of distributed editors based on a family of distributed data structure replicated on several sites called Conflict-free Replicated Data Type (CRDT for short) appeared recently. This paper considers a CRDT that represents a distributed sequence of basic elements that can be lines, words or characters (sequence CRDT). The possible operations on such a sequence are the insertion and deletion of elements. Compared to the state of the art, this approach is more decentralized and better scales in terms of the number of participants. However, its space complexity is linear with respect to the total number of inserts and the insertion points in the document. This makes the overall performance of such editors dependent on the editing behaviour of users. This paper proposes and models LSeq, an adaptive allocation strategy for a sequence CRDT. LSeq achieves in the average a sub-linear spatial complexity whatever is the editing behaviour. Series of experiments validate LSeq showing that it outperforms existing approaches.

Concurrency Effects Over Variable-size Identifiers in Distributed Collaborative Editing
– Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils

Workshop paper – Document Changes'13 : Modeling, Detection, Storage and Visualization (DChanges'13), 2013

Abstract : Distributed collaborative editors such as Google Docs or Etherpad allow to distribute the work across time, space and organizations. In this paper, we focus on distributed collaborative editors based on the Conflict-free Replicated Data Type approach (CRDT). CRDTs encompass a set of well-known data types (sets, graphs, sequences, etc.). CRDTs for sequences model a document as a set of elements (character, line, paragraph, etc.) with unique identifiers, providing two commutative update operations : insert and delete. The identifiers of elements can be either of fixed-size or variable-size. Recently, a strategy for assigning variable-size identifiers called LSeq has been proposed for CRDTs for sequences. LSeq lowers the space complexity of variable-size identifiers CRDTs from linear to sub-linear. While experiments show that it works locally, it fails to provide this bound with multiple users and latency. In this paper, we propose h-LSeq, an improvement of LSeq that preserves its space complexity among all collaborators, regardless of the latency. Ultimately, this improvement allows to safely build distributed collaborative editors based on CRDTs. We validate our approach with simulations involving latency and multiple users.

2

Une structure de séquences réparties à large échelle

Sommaire

Les éditeurs Web tels que Google Docs [40] ou ShareLaTeX[81] ont fortement contribué à l'adoption des éditeurs collaboratifs par le grand public [68]. Des millions d'utilisateurs partagent et rédigent leurs documents en temps réel directement dans leurs navigateurs Web. Cependant un serveur central appartenant à un fournisseur de services sert d'intermédiaire à l'édition. Cela soulève des problèmes en termes de confidentialité, de censure, de propriété et d'intelligence économique [19, 37, 82]. À cela s'ajoutent des problèmes de tolérance aux pannes et de passage à l'échelle, notamment en ce qui concerne le nombre d'utilisateurs. Bien que les petits groupes de collaborateurs soient la cible principale de ce genre d'application, certains événements de plus ample dimension tels que les cours en ligne

ouverts et massifs (*MOOC*) [15] nécessitent de supporter des groupes à la fois plus larges et plus dynamiques. En effet, si un cours commence avec quelques milliers d'étudiants prenant des notes, sa population diminue en fonction de l'intérêt qu'il suscite, avant de récupérer certains étudiants désireux de réussir leurs examens. Google Docs gère les groupes de grande taille. En revanche, seuls les 50 premiers collaborateurs ont le droit de modifier le document en temps réel, les suivants voient leur accès limité à la simple lecture. Selon nous, même si seulement un petit ensemble parmi les millions de collaborateurs est réellement en train d'écrire, n'importe quel participant de la session d'édition devrait pouvoir lire et écrire lorsqu'il le souhaite.

La décentralisation des éditeurs collaboratifs constitue une étape nécessaire vers une solution aux problèmes liés à la confidentialité : un document doit réellement appartenir à ceux qui le rédigent. Cependant, les problèmes de passage à l'échelle demeurent. Résoudre ces problèmes revient à trouver un bon compromis entre la complexité en communication, en espace et en temps de l'éditeur. En particulier, obtenir une complexité en communication sous-linéaire par rapport au nombre de participants est crucial pour gérer les groupes de grande envergure.

Afin d'augmenter la réactivité aux changements et la disponibilité des documents, les éditeurs temps réel actuels emploient le schéma de réplication optimiste [23, 55, 88, 95] de séquences – les documents étant simplement représentés par des séquences de caractères. En tant que tel, chaque éditeur

(i) héberge une copie locale du document. L'évolution de la mémoire prise par cette copie constitue la complexité spatiale de l'approche ;

(ii) modifie directement cette copie locale. L'évolution des performances des opérations de modification constitue la complexité temporelle à la génération de l'approche ;

(iii) propage le résultat de ses opérations à tous les éditeurs impliqués dans la rédaction du document. L'évolution du nombre et de la taille des messages disséminés constitue la complexité en communication de l'éditeur ;

(iv) reçoit les messages et intègre leur contenu sur sa propre réplique. L'évolution des performances de l'intégration de ces opérations de modification constitue la complexité temporelle à l'intégration de l'approche.

La réplication optimiste fait l'hypothèse d'une dissémination fiable des messages, i.e., tous les éditeurs reçoivent l'intégralité des messages. Le système est correct si les répliques intégrant un même ensemble d'opérations convergent vers un état équivalent, i.e., les collaborateurs lisent un même document [9, 90].

Les algorithmes décentralisés appartenant aux transformées opérationnelles (*OT*) [94, 96] transportent dans chaque message un vecteur d'état ou de contexte dans le but de détec-

ter les opérations concurrentes. La taille de ces vecteurs croît linéairement avec le nombre de membres ayant jamais participé à l'édition du document. Ainsi, ces approches sont efficaces pour les petits groupes d'utilisateurs mais ne sont pas adaptées aux groupes de plus large dimension, plus dynamiques, et sujet aux aléas du réseau – notamment sur la latence.

Les structures de données répliquées sans conflits (*CRDTs*) [16, 89, 90], contrairement aux approches OT, ne payent pas le prix de la détection de la concurrence entre les opérations. Toutefois, ils transportent des identifiants uniques et immuables pour chaque opération propagée sur le réseau. La taille de ces identifiants a un impact direct sur le trafic généré. Deux classes de CRDTs conçues pour les séquences existent :

Pierre tombales [4, 8, 21, 42, 80, 86, 104, 109, 110]. Les CRDTs tels que WOOT [80] transportent un identifiant de taille constante. Toutefois, les éléments supprimés sont simplement cachés à l'utilisateur tout en restant présents dans la structure. La consommation en espace de la réplique croît inexorablement. Pour réellement détruire les éléments supprimés, l'exécution périodique d'un ramasse-miettes réparti [1] est nécessaire. Malheureusement, cela ne passe pas à l'échelle [1].

Identifiants de taille variable [7, 84, 105]. Les CRDTs tels que Logoot [105] ne nécessitent pas de pierres tombales mais transportent des identifiants de taille variable. En fonction de la stratégie d'allocation, ces identifiants peuvent croître linéairement par rapport au nombre d'insertions effectuées dans le document. Puisque chaque identifiant doit être envoyé à l'ensemble des répliques, la complexité en communication s'en trouve directement impactée. Pour équilibrer la structure *a posteriori*, l'exécution périodique d'un mécanisme de relocalisation [60] est nécessaire. Malheureusement, dans notre contexte, atteindre un tel consensus s'avère très coûteux [72].

Afin d'éviter tout protocole additionnel de relocalisation des identifiants, comment allouer des identifiants de taille variable dont la taille soit directement sous-linéaire ?

Ce chapitre présente LSeq [75, 76], une fonction d'allocation d'identifiants dont la taille croît de manière polylogarithmique par rapport au nombre d'insertions dans la séquence. À ce titre, elle évite l'utilisation de protocoles de relocalisation.

La section 2.1 présente l'état de l'art. Elle introduit le schéma de réplication optimiste des séquences avant d'en détailler les approches ainsi que les limites auxquelles elles sont assujetties. La section 2.2 définit le problème scientifique à résoudre. La section 2.3 présente LSeq, une approche basée sur les opérations commutatives dont la complexité est sous-linéaire. La section 2.4 s'attache à démontrer cette complexité ainsi que les conditions sous lesquelles elle s'applique. La section 2.5 valide LSeq au travers de simulations. La section 2.6 conclut ce chapitre.

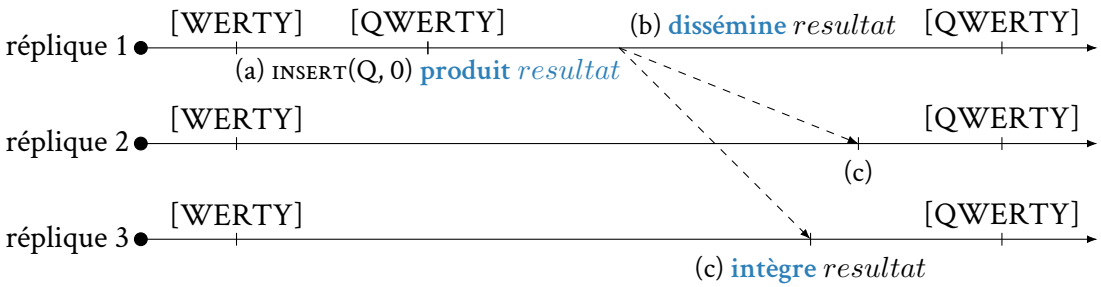


FIGURE 2.1 – Les 3 étapes de la réplication optimiste.

2.1 État de l'art

Cette section commence par décrire le schéma de réplication optimiste de séquences – un document pouvant être représenté par une séquence de caractères (cf. §2.1.1). Ensuite, cette section passe en revue deux familles d'approches appartenant à la réplication optimiste des séquences : les transformées opérationnelles (cf. §2.1.2) et les structures de données répliquées sans conflits (cf. §2.1.3). Cette section s'attache particulièrement à cette dernière famille et en détaille les approches.

2.1.1 Réplication optimiste de séquences

La réplication optimiste [23, 52, 55, 88] de séquence est un paradigme de réplication où chaque éditeur possède une réplique locale du document. Un document est toujours disponible et réactif aux changements effectués. En effet, comme l'illustre la figure 2.1, (a) une opération de modification est directement exécutée sur la réplique locale de la séquence. La modification produit un résultat qui (b) est disséminé aux autres serveurs hébergeant une réplique. (c) Le serveur exécute – ou *intègre* – le résultat reçu.

La réplication optimiste fait l'hypothèse d'une dissémination fiable où tous les serveurs reçoivent toutes les modifications afin que toutes les répliques convergent vers un état équivalent. Ainsi, après réception et intégration du $resultat$, les trois répliques de la figure 2.1 contiennent la même séquence `QWERTY`.

Sun et al. [95] déclarent que l'édition collaborative temps réel nécessite un système préservant les trois propriétés CCI :

Convergence. Les répliques ayant reçues les mêmes opérations convergent vers un état identique. Par exemple, la figure 2.1 montre que les répliques peuvent avoir des états temporairement divergent. En revanche, après réception et intégration du résultat de l'opération d'insertion, les trois répliques convergent vers une séquence identique.

La convergence fait l'objet de critère de cohérence à part entière tels que la cohérence à

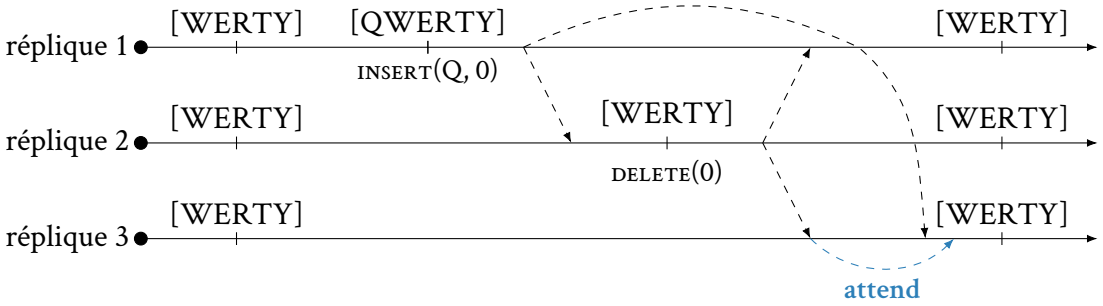


FIGURE 2.2 – Exemple de respect des relations causales entre deux opérations.

terme [9], ou la cohérence forte à terme [90]. Ces critères de cohérence sont très faibles mais ont beaucoup de succès ces dernières années avec, par exemple, les bases de données réparties [6, 10, 31, 46].

Causalité. Si une opération précède une autre opération [56], alors l'intégration de cette première opération précède l'intégration de la seconde opération. La causalité contraint l'ordre d'intégration des opérations. Ainsi, une opération dépendante d'une autre opération effectuée avant se trouve forcément intégrée à sa suite. La figure 2.2 montre un exemple où, sur la réplique 3, l'intégration de l'opération de suppression doit attendre l'arrivée de l'opération d'insertion pour supprimer le caractère `Q`.

Intention. L'effet observé sur le document lors de la génération d'une opération doit être également observé lors de son intégration. Dans l'exemple de la figure 2.2, la cible originelle de la suppression sur la réplique 2 est le caractère `Q`. Par conséquent, ce caractère doit être supprimé lors de l'intégration de l'opération, et nul autre caractère.

L'intention des opérations d'une séquence demeure difficile à formaliser dans le cas général. L'effet d'une opération doit respecter le plus possible sa spécification séquentielle [12]. La spécification séquentielle d'une séquence inclut deux opérations : « insérer l'élément e à la position i dans la séquence » et « supprimer l'élément à la position i dans la séquence ». L'intention semble donc liée à la position des éléments dans la séquence. Contre cette intuition, nous soutenons qu'une séquence se définie plus simplement par un ordre dense sur ses éléments : les éléments sont ordonnés et il est toujours possible d'insérer un élément entre deux autres éléments. Les opérations usuelles d'insertion et de suppression se traduisent par une transformation de position dans la séquence vers un ordre dense.

Définition 1 (Spécification séquentielle d'une séquence). *Soit une série d'opérations H produisant la séquence $s(H) = \{p_1, p_2 \dots p_k\}$ avec $p_{1..k} \in \mathcal{P}$ où $\mathcal{P}$ est un ensemble muni d'un ordre dense $(\mathcal{P}, <_\mathcal{P})$ tel que :*
$\forall p \in \mathcal{P}, p_\vdash <_\mathcal{P} p <_\mathcal{P} p_\dashv$ *et* $p_\vdash <_\mathcal{P} p_1 <_\mathcal{P} p_2 <_\mathcal{P} \dots <_\mathcal{P} p_k <_\mathcal{P} p_\dashv$.

L'insertion d'un élément e en position i dans la séquence $s(H)$ est définie de la façon suivante :

$$s(H \cup \textsc{insert}(i,\ e)) \rightarrow s(H) \cup \begin{cases} \{p,\ p_{\vdash} <_{\mathcal{P}} p <_{\mathcal{P}} p_{\dashv}\} & i = 0 \wedge |s(H)| = 0 \\ \{p,\ p_{\vdash} <_{\mathcal{P}} p <_{\mathcal{P}} p_1\} & i = 0 \wedge |s(H)| > 0 \\ \{p,\ p_k <_{\mathcal{P}} p <_{\mathcal{P}} p_{\dashv}\} & i = k \\ \{p,\ p_i <_{\mathcal{P}} p <_{\mathcal{P}} p_{i+1}\} & sinon \end{cases} \tag{2.1}$$

La suppression de l'élément en position i dans la séquence $s(H)$ est définie de la façon suivante :

$$s(H \cup \textsc{delete}(i)) \rightarrow s(H) \setminus \{p_i\} \tag{2.2}$$

Nous distinguons la complexité spatiale de la réplique, et la complexité en communication sur les messages, la complexité temporelle d'une opération exécutée localement, la complexité temporelle de l'intégration d'une opération reçue.

Parmi ces complexités, la complexité en communication est la plus critique et doit être sous-linaire pour passer à l'échelle. En ce qui concerne les complexités temporelles, il est plus important d'améliorer l'intégration que la génération car chaque opération locale effectuée conduit à N intégrations, où N est le nombre de répliques. Les opérations étant atomiques, leur exécution est bloquante vis-à-vis des autres opérations. Un temps d'exécution trop long constitue un danger pour l'édition en temps réel.

Le reste de cette section présente les approches appartenant à la réplication optimiste de séquences : les approches à transformées opérationnelles (cf. §2.1.2) et les approches à structure de données proposant des opérations commutatives (cf. §2.1.3).

2.1.2 Transformées opérationnelles

Les approches basées sur les transformées opérationnelles (OT) [94, 96] sont les plus anciennes. Les opérations locales d'insertion et de suppression proposées possèdent une signature correspondant exactement à la signature commune des séquences : $\textsc{insert}(element, position)$ et $\textsc{delete}(position)$. Lors de la réception d'une opération, ses arguments sont ajustés afin qu'ils s'appliquent à l'état courant de la réplique malgré les opérations effectuées et intégrées en concurrence. Cette phase d'intégration nécessite l'examen des opérations concurrentes afin d'en compenser les effets sur la séquence.

La figure 2.3 illustre le principe de fonctionnement des approches basées sur OT sur un scénario impliquant une séquence répliquée. Dans cet exemple, les répliques sont toutes initialisées avec la séquence `WERTY`. Ensuite, tandis que la première réplique insère le caractère `Q` en tête de séquence pour obtenir `QWERTY`, la troisième réplique supprime son premier caractère correspondant au `W`. Lorsque la réplique 1 reçoit cette dernière opération de suppression, elle est interprétée comme une opération dont le contexte d'exécution n'avait pas

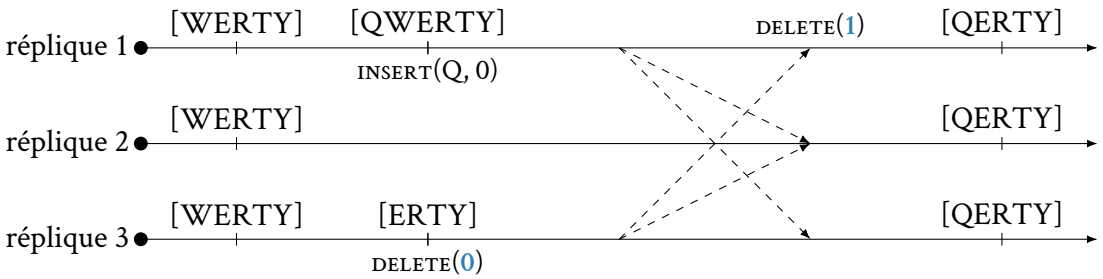


FIGURE 2.3 – Exemple de scénario impliquant des transformées opérationnelles. L'opération de suppression du premier caractère sur la réplique 3 est transformée afin de supprimer le second caractère sur les autres répliques.

encore intégré l'insertion du caractère `Q`. Le décalage d'une position vers la droite de chaque caractère n'avait donc pas encore été intégré. Par conséquent, l'argument de l'opération voit sa cible changée au second caractère : `W`. Réciproquement, la réplique 3 intègre l'insertion de la réplique 1 : l'insertion et la suppression sont détectées concurrentes, toutefois aucune transformation n'est nécessaire. À terme, les répliques convergent vers la séquence `QERTY`.

Dans les approches OT décentralisées [96], chaque client est aussi un serveur hébergeant une réplique de la séquence. Deux problèmes apparaissent :

Coût de la détection. Chacune de ces entités doit être en mesure d'effectuer les transformations d'elle-même. Parmi les pré-requis à cette tâche figure le mécanisme de détection de concurrence. En effet, retrouver le contexte d'exécution revient à transformer l'opération reçue contre toutes celles qui ont été intégrées sans en avoir connaissance. Cependant, chaque message doit transporter un vecteur d'horloges (*vector clock*) [56] ou de contexte pour chaque opération. Par conséquent, la complexité en communication est au minimum linéaire par rapport au nombre de répliques. De plus, ces vecteurs et leur opération associée sont ensuite sauvegardés dans un historique. La complexité spatiale est au minimum linéaire selon les deux dimensions : nombre de répliques et nombre d'opérations.

Répartition des temps d'exécution. La génération d'une opération s'exécute en temps constant. Cependant, son intégration s'exécute en temps quadratique par rapport au nombre d'opérations concurrentes à l'opération intégrée. Cette répartition des coûts d'opération est malheureuse car 1 opération locale efficace correspond à N exécutions distantes potentiellement lentes, où N est le nombre de répliques recevant l'opération.

Pour ces raisons, les approches OT décentralisées ne passent pas à l'échelle. A contrario, dans un environnement bien maîtrisé, où les opérations arrivent très rapidement à un groupe raisonnable de participants, ces approches décentralisées sont extrêmement efficaces [65].

2.1.3 Structure de données répliquée sans conflits

Les structures de données répliquées sans conflits (*CRDTs*[1]) [89, 90] appartiennent au schéma de réplication optimiste. Ces approches sont basées sur des structures de données abstraites fournissant des opérations dont les résultats commutent par nature. Les répliques convergent donc vers un état identique même en cas de concurrence. Contrairement aux approches OT, les opérations concurrentes n'ont pas besoin d'être détectées. Les CRDTs s'affranchissent donc de la nécessité de joindre un vecteur d'horloges à chaque opération. Sans cette contrainte, les CRDTs ont l'opportunité de passer à l'échelle. Contrairement aux approches OT, la signature des opérations est différente des séquences « classiques » et correspond à l'intention telles que nous la définissons où une séquence est un ensemble d'éléments muni d'un ordre dense : les éléments sont ordonnés et il est toujours possible d'insérer un élément entre deux autres éléments. Par exemple, « insérer l'élément e à la position i » devient « insérer l'élément e entre l'élément en position i et l'élément en position $i + 1$ ».

Pour fonctionner , les CRDTs pour séquences surchargent chaque élément d'une métadonnée nommée identifiant. Les identifiants, uniques et immuables, permettent d'assurer un ordre total (convergence) sur les éléments tout en respectant un ordre dense (intention) établi lors de l'exécution locale de l'insertion. Par exemple, si l'élément e est inséré entre l'élément p et l'élément q, alors il n'existe aucune réplique où l'intégration de cette opération résulterait en une séquence où e n'est pas placé – directement ou non – entre p et q. Si p ou q n'existe(nt) plus car supprimé(s), alors l'intégration de e doit tout de même respecter l'ordre dense établi lors de l'insertion de p et q. Par exemple, si o précède p dans la séquence mais que p est supprimé, alors o précède – directement ou non – l'élément e.

Selon la manière dont les identifiants sont générés, nous distinguons deux types d'approches : les approches recourant à des marqueurs nommés « pierre tombales » (cf. §2.1.3.a) ; et les approches générant des identifiants dont la taille varie (cf. §2.1.3.b).

2.1.3.a Pierres tombales

Une « pierre tombale » est une marque laissée après la suppression d'un élément indiquant qu'un jour, celui-ci a existé à cet emplacement. Ces marques sont cachées à l'utilisateur : les caractères disparaissent du document mais existent toujours dans la structure répliquée.

WOOT [80] : WOOT est le premier représentant historique des CRDTs pour séquences suivi par deux extensions **WOOTO [104]** et **WOOTH [4]**. Dans cette approche chaque identifiant fait référence aux identifiants voisins à l'insertion. Lorsqu'ils sont rassemblés, les identifiants peuvent être ordonnés grâce à un diagramme de Hasse. Toutefois, cet ordonnancement requiert des deux bornes adjacentes qu'elles soient (i) déjà intégrées et (ii) toujours présentes.

[1] *Conflict-free Replicated Data Types*

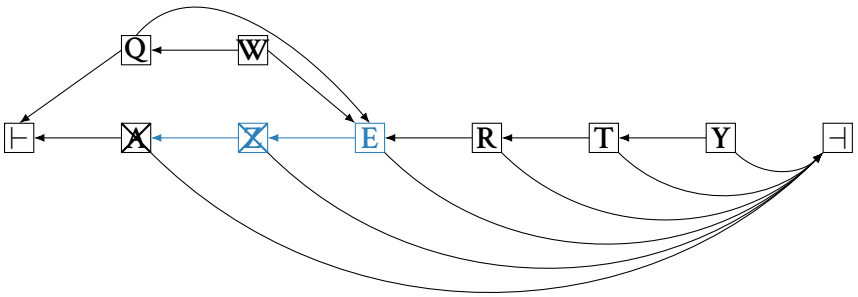


FIGURE 2.4 – Le diagramme de Hasse du modèle WOOT représentant la séquence QWERTY. Bien que supprimé, le caractère Z est indispensable au bon ordonnancement de la séquence.

La figure 2.4 illustre la nécessité de conserver les pierres tombales. Elle montre le diagramme de Hasse généré lors du scénario suivant. Tout d'abord, un utilisateur écrit AZERTY. Ensuite, les deux premiers caractères sont supprimés afin d'être remplacés par les caractères QW. La séquence finale est QWERTY. Toutefois, les identifiants ne sont pas modifiables, et l'identifiant du caractère E référence l'identifiant de Z, lui-même référençant l'identifiant de A. Par conséquent, supprimer complètement les identifiants de A et/ou de Z revient à rendre l'identifiant de E non positionnable, et tout ceux qui en dépendent par transitivité.

Même si le document peut apparaître vide, la structure répliquée contient tous les éléments ayant été insérés depuis sa création. Cette croissance monotone de la structure impacte négativement l'intégration des opérations qui devient moins efficace. Du reste, la complexité en communication est optimale puisque seul un identifiant de taille constante doit être envoyé par opération.

Causal tree [42] : Cette approche caractérise explicitement les relations causales grâce à une représentation sous forme d'arbre. Ainsi, chaque opération est accompagnée de l'identifiant de la dernière opération observée. En parcourant l'arbre et en appliquant les opérations, la séquence peut être retrouvée. Toutefois, les identifiants sont des vecteurs d'horloges dont la taille croît linéairement avec le nombre de participants. De plus, il est nécessaire de conserver tous les nœuds de cet arbre causal au cas où une opération y ferait référence. Cette approche garantie la réception causale comme le prescrit la cohérence CCI [95]. Cependant, elle n'est pas adaptée aux contextes où le groupe d'éditeurs est grand.

Partial persistent sequence [109] : Cette approche définit les identifiants dans l'ensemble des nombres rationnels auxquels est ajoutée une limite quant à leur précision. Hélas, cette limite contraint la taille maximale que peut atteindre un document. Sans cette troncature, l'approche serait susceptible d'appartenir à l'autre famille de CRDTs pour séquence.

Replicated growable array [86] : Cette structure représente la séquence sous forme de liste supportant les opérations concurrentes. Une table de hachage apporte un accès rapide aux éléments grâce à leurs identifiants. Les éléments incluent une référence au voisin qu'ils précèdent lors de leur insertion. Toutefois, pour ne jamais briser la chaîne ainsi construite, les éléments supprimés sont cachés et restent présents dans la structure. Une variante sous forme d'arbre a récemment été proposée [8]. Tout comme pour WOOT, la mémoire locale consommée par la réplique est monotone croissante. Les performances des opérations se dégradent progressivement à chaque insertion.

String-wise [110] : Cette approche cible principalement les chaînes de caractères pouvant être subdivisées lors d'opérations jusqu'à devenir une série de caractères. Les identifiants référencent alors les chaînes adjacentes à l'insertion ainsi que les autres éléments de la chaînes si subdivision il y a. Le nombre d'identifiants générés s'en trouve diminué. Cependant, de la même manière que pour les approches précédentes, les références empêchent à la structure répliquée de se débarasser des éléments supprimés. La complexité spatiale de l'approche est monotone croissante et linéaire par rapport au nombre d'insertions et de subdivisions. Les performances des opérations en sont impactées négativement.

Pour préserver l'ordre dense défini lors de l'exécution locale de l'insertion, toutes ces approches génèrent des identifiants qui référencent au moins l'un des identifiants adjacent. Il devient impossible de supprimer un élément de la structure répliquée sans mettre en danger l'ordre dense à préserver. Ainsi, les suppressions conservent tous les éléments, qu'ils soient supprimés ou non du document. Un document peut être vide bien que la séquence répliquée possède des milliers d'éléments cachés. La mémoire consommée par ces approches croît au moins linéairement par rapport au nombre d'insertions faites dans la séquence. Plus problématique : cette croissance est monotone. Les pierres tombales dégradent à jamais les performances de l'intégration des opérations où l'ordre des éléments doit être retrouvé.

Purger la structure de données des éléments cachés est une solution potentielle aux dégradations de performances. Un mécanisme de ramasse-miettes réparti [1] permet de nettoyer une structure de données en vidant de la mémoire les objets qui ne sont plus accessibles par le programme, ni localement ni à distance. Ainsi, supprimer réellement un élément de la séquence revient à s'interroger : « Est-ce que (i) toutes les répliques ont supprimé l'élément et (ii) tous les éléments référençant l'élément supprimé ont été intégrés localement ? » Cela va sans dire qu'il est difficile d'apporter une réponse à ces deux questions. D'autant plus lorsque les répliques ne sont pas perpétuellement accessibles. Corenebula [60] propose de contraindre la topologie du réseau afin de rendre la prise de décision possible. Ainsi, un cœur décisionnel prend en charge les choix de suppression réelle des objets. Ce cœur décisionnel est restreint à un sous-ensemble de membres du réseaux toujours accessibles. Les décisions peuvent alors être prises de manière fiable. Le reste des participants se conforme à ces décisions au risque de perdre certaines de leurs modifications.

Cette re-centralisation expose le système aux problèmes de confidentialité, de censure, de propriété, de passage à l'échelle et de tolérance aux défaillances.

Un autre famille de CRDTs existe pour le type séquence dont le bon fonctionnement ne nécessite pas de référencer directement d'autres identifiants. Cela évite l'usage des pierres tombales mais pose le problème de la complexité spatiale des identifiants.

2.1.3.b Identifiants de taille variable

Contrairement aux approches basées sur les pierres tombales, certains CRDTs génèrent des identifiants ne référençant pas d'autres identifiants. Par conséquent, ces identifiants sont indépendants : leur intégration ne nécessite pas l'intégration d'autres identifiants. Ces identifiants ont une taille variable définie à la génération [7, 84, 105]. Ainsi, les identifiants sont toujours uniques et immuables une fois générés, mais leur structure est une liste de valeurs encodant la position relative de l'élément par rapport aux positions relatives des éléments adjacents à son insertion. Ces identifiants sont directement créés dans un ensemble muni d'un ordre total pour assurer la convergence, et muni d'un ordre dense pour assurer l'intention. À ce titre, ces approches constituent une implémentation directe de notre spécification séquentielle des séquences.

Logoot [103, 105, 106] : Ce CRDT représente la séquence sous forme de liste d'identifiants. Chacun des identifiants est une liste de triples notée $id = [t_1, t_2, \ldots, t_k]$ où $t_k = \langle p_k, s_k, c_k \rangle$ où p_k est un entier positif, s_k est un identifiant unique de site, et c_k est un compteur local au site s_k. Un ordre lexicographique est utilisé pour préserver l'ordre parmi les éléments :

$$\begin{aligned}
t_i < t_j &\iff (p_i < p_j) \vee \\
&\quad ((p_i = p_j) \wedge (s_i < s_j)) \vee \\
&\quad ((p_i = p_j) \wedge (s_i = s_j) \wedge (c_i < c_j)) \\
t_i = t_j &\iff \neg(t_i < t_j) \wedge \neg(t_j < t_i) \\
id_i <_{\mathcal{I}} id_j &\iff \exists (m > 0)(\forall n < m),\ (t_n^i = t_n^j) \wedge (t_m^i < t_m^j) \\
id_i =_{\mathcal{I}} id_j &\iff \forall m,\ t_m^i = t_m^j
\end{aligned}$$

L'identifiant unique de site est obtenu lors de la création de la réplique. Le compteur est incrémenté à chaque insertion locale dans la séquence. Ne reste comme seul facteur de choix à disposition de Logoot que l'entier p présent dans chaque triple. Lors de la $x^{\text{ème}}$ insertion locale par le site s_y, les identifiants adjacents à la position d'insertion sont récupérés. Ainsi l'identifiant id du nouvel élément inséré en position i doit respecter $id_i <_{\mathcal{I}} id <_{\mathcal{I}} id_{i+1}$. Par transitivité $\ldots id_{i-1} <_{\mathcal{I}} id_i <_{\mathcal{I}} id <_{\mathcal{I}} id_{i+1} <_{\mathcal{I}} id_{i+2} \ldots$. Prenons pour exemple, $id_i = [\langle 12, s_1, c_1 \rangle]$ et $id_{i+1} = [\langle 12, s_1, c_1 \rangle . \langle 1337, s_2, c_2 \rangle]$. Pour choisir les valeurs successives de p du nouvel identifiant, Logoot commence par examiner chaque niveau des identifiants adjacents afin de connaître la taille du nouvel identifiant. Ici, les identifiants

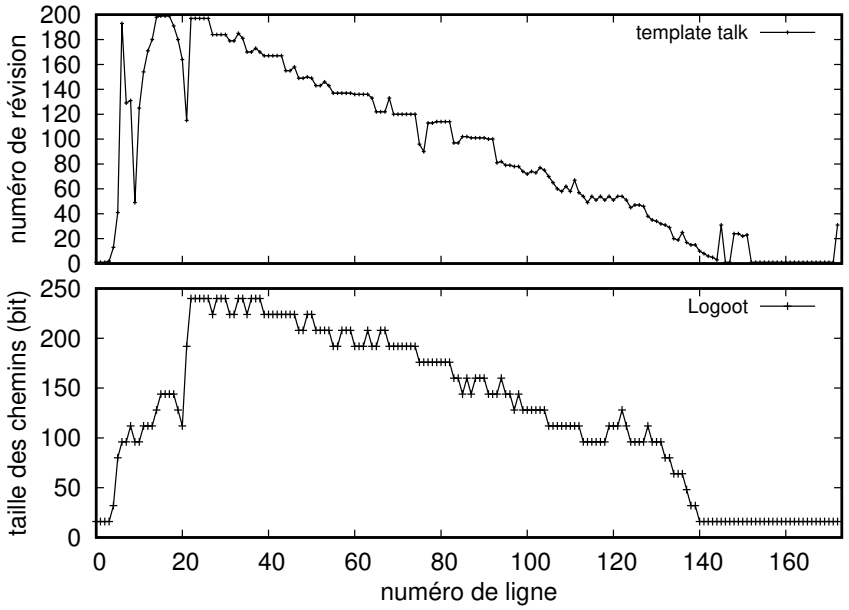


FIGURE 2.5 – Taille des chemins alloués par Logoot sur un document principalement édité en tête. L'axe des abscisses montre le numéro de la ligne dans le document. L'axe des ordonnées de la partie haute de la figure montre la révision à laquelle a été ajoutée la ligne. L'axe des ordonnées de la partie basse de la figure montre la taille du chemin alloué à une ligne.

commencent tous les deux avec l'entier 12. Puisqu'entre 12 et 12 aucune valeur entière n'est disponible, Logoot doit chercher plus loin dans les identifiants. L'identifiant précédant la position d'insertion ne possède pas de second triple ce qui correspond implicitement à $\langle 0,\ -\infty,\ -\infty\rangle$. Entre 0 et 1337 de nombreuses valeurs sont disponibles. Logoot doit choisir parmi toutes ces valeurs laquelle constituera définitivement le nouvel identifiant. Ce choix impactera également l'allocation des identifiants à venir. Des observations effectuées sur un corpus de documents Wikipedia [101] suggère qu'allouer un identifiant proche de l'identifiant précédent est une meilleure stratégie afin de conserver des identifiants de petite taille. Ici, Logoot choisit donc une valeurs proche de 0 et compose le nouvel identifiant en copiant une partie des données présentes dans les identifiants adjacents. Par exemple, $id = [\langle 12,\ s_1,\ c_1\rangle.\langle 10,\ s_y, x\rangle]$. En utilisant l'ordre lexicographique défini auparavant, nous obtenons bien $id_i <_{\mathcal{I}} id <_{\mathcal{I}} id_{i+1}$.

Logoot favorise, par son choix d'entier proche de l'identifiant précédent la position d'insertion, les comportements d'édition où les éléments sont insérés de gauche à droite. Toutefois, lorsque le comportement d'édition va à l'encontre de cette stratégie, la taille des identifiants peut augmenter extrêmement rapidement. La figure 2.5 met en évidence cette crois-

sance. La taille des chemins (i.e. les identifiants uniques de site et compteurs sont ignorés) augmente par pallier. De nombreuses séries d'insertions en tête se traduisent par l'ajout d'un niveau dans les chemins. Sur un petit document de moins de 200 lignes, un chemin alloué par Logoot atteint 16 concaténations. Ici, chacune de ces concaténations pèse 16 bits. La complexité spatiale de chaque identifiant est linéaire par rapport au nombre d'insertions dans le document, et puisque chaque identifiant est disséminé aux autres répliques, la complexité en communication adopte cette progression. De plus, les valeurs choisies par Logoot dans les identifiants sont toujours comprises entre 0 et 2^c où c est une constante ($c = 16$ dans la figure 2.5). Cette constance implique que, même dans le cadre du comportement d'édition attendu, la complexité est linéaire – bien que fortement atténuée. Là encore, le trafic généré en souffre directement.

La représentation de la séquence sous forme de liste possède l'avantage de fournir un accès instantané à ses éléments. Ainsi, la traduction de « insérer l'élément e à la position i » à « insérer l'élément e ente l'identifiant id_i de l'élément en position $i+1$ et l'identifiant id_i de l'élément en position i » s'effectue en temps constant. Cependant, cette rapidité d'accès est seulement possible au détriment d'une consommation en espace élevée : Logoot ne profite pas de la redondance d'information sur les triples. La complexité spatiale de la structure est quadratique par rapport au nombre d'insertions effectuées dans la séquence.

Une extension nommée **LogootSplit** [7] étend Logoot lui offrant la possibilité d'adapter la granularité de l'élément ciblé. Cette extension prend pour granularité la chaîne de caractères avec la liberté de la scinder si besoin est. Le nombre d'identifiants à allouer s'en trouve diminué, et par conséquent, le trafic généré également. Cette amélioration est orthogonale et s'applique à tous les CRDTs générant des identifiants de taille variable.

Treedoc [60, 84] : Cette approche représente le document sous forme d'arbre binaire avec parcours infixe. Lorsqu'il n'y a pas de concurrence, chaque nœud de l'arbre possède au plus un élément et au plus 2 fils. Les arêtes de l'arbre sont étiquetées par la valeur binaire 0 ou 1. Si un nœud possède un élément et deux fils, tous les éléments du sous-arbre accessible par l'arête dont l'étiquette est 0 se trouvent à gauche de ce premier élément ; tous les éléments du sous-arbre accessible par l'arête dont l'étiquette est 1 se trouve à droite de ce premier élément. Du fait de la concurrence, chaque nœud peut aussi devenir un super-nœud accueillant plusieurs nœuds. De plus, chaque nœud est décoré d'un identifiant unique de site et d'un compteur local à ce site. Ainsi, les nœuds contenus dans chaque super-nœud suivent eux aussi un ordre total. Un identifiant est donc une liste de triples $id = [t_1.t_2 \dots t_k]$ où k est un entier, et $t_k = \langle b_k,\ s_k,\ c_k \rangle$ où b_k est une valeur binaire, s_k est un identifiant unique de site, et c_k est un compteur local au site s_k. Prenons par exemple les éléments adjacents $id_i = [\langle 0,\ s_1,\ c_1 \rangle.\langle 1,\ s_1,\ c_2 \rangle]$ et $id_{i+1} = [\langle 0,\ s_1,\ c_1 \rangle.\langle 1,\ s_1,\ c_2 \rangle.\langle 1,\ s_2,\ c_3 \rangle]$. Lors de la $x^{\text{ème}}$ insertion locale par le site s_y, Treedoc examine les identifiants adjacents à la position d'insertion. Treedoc commence par examiner si un identifiant est l'ancêtre direct de l'autre, i.e., la liste de triples d'un identifiant est incluse dans la liste de triples de l'autre identifiant.

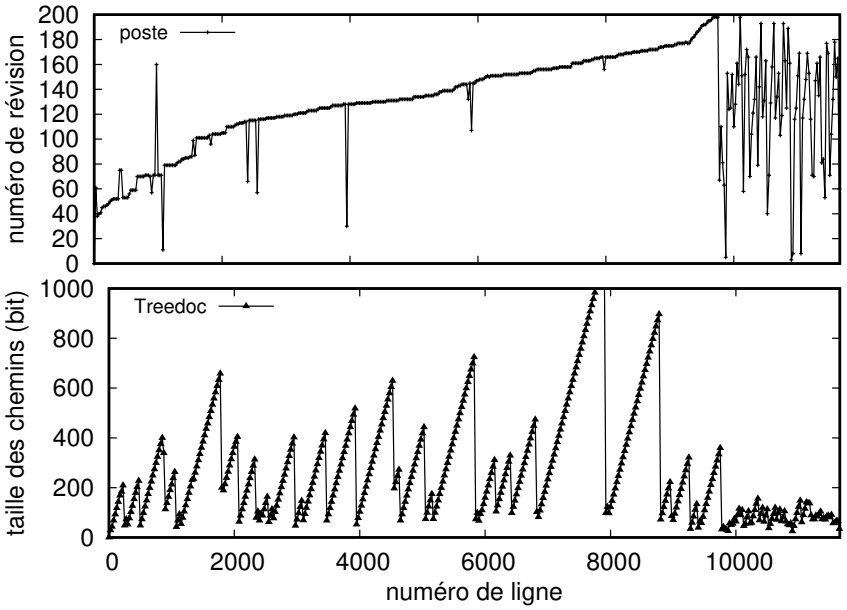


FIGURE 2.6 – Taille des chemins alloués par Treedoc sur un grand document principalement édité en fin. L'axe des abscisses montre le numéro de la ligne dans le document. L'axe des ordonnées de la partie haute de la figure montre la révision à laquelle a été ajoutée la ligne. L'axe des ordonnées de la partie basse de la figure montre la taille du chemin alloué à une ligne.

Par exemple, id_i est l'ancêtre de id_{i+1}. Dans ce cas, Treedoc copie l'identifiant du descendant et le suffixe par $\langle 1,\ s_y,\ x\rangle$ si le descendant précède la position d'insertion ; $\langle 0,\ s_y,\ x\rangle$ dans le cas contraire. Si aucun des identifiants n'est l'ancêtre de l'autre, Treedoc choisit de copier l'identifiant précédant la position d'insertion et lui suffixe le triple $\langle 1,\ s_y,\ x\rangle$.

La figure 2.6 met en évidence la croissance rapide des chemins (i.e. les identifiants uniques de site et compteurs sont ignorés) alloués par Treedoc. Un série d'insertions se succédant conduit à des identifiants de plus de 1k bits. Tout comme l'approche Logoot, les observations effectuées sur un corpus de documents Wikipédia montrent l'importance d'une stratégie d'allocation gérant l'édition de gauche à droite. Elles conduisent à proposer une heuristique selon laquelle des nœuds de l'arbre sont virtuellement créés en prévoyance des éditions à venir. Toutefois, de manière identique à Logoot, lorsque le comportement d'édition va à l'encontre de cette stratégie, les identifiants sont de taille linéaire par rapport à la taille du document.

La représentation de la séquence sous forme d'arbre possède l'avantage de factoriser les nombreux triples communs composant les identifiants Treedoc. Cette forme compacte de structure a le défaut de ne pas permettre d'accès direct aux identifiants. Retrouver les iden-

tifiants adjacents à la position d'insertion requiert un parcours de l'arbre. De plus, cette représentation compacte n'a pas d'incidence sur les identifiants eux-mêmes. Puisque chaque identifiant doit être disséminé individuellement aux autres répliques, le trafic généré demeure inchangé par ce choix de structure.

Les CRDTs fonctionnant avec des identifiants de taille variable n'utilisent pas de pierres tombales lors de la suppression d'éléments. Par conséquent, la mémoire consommée par ces approches n'est pas strictement croissante. Un document vide signifie que la séquence répliquée est également vide. En revanche, les identifiants peuvent croître de manière linéaire par rapport au nombre d'insertions dans le document. Malheureusement, cette progression se répercute sur la complexité en communication. De même, cette progression se répercute sur les performances à l'intégration des opérations d'insertion.

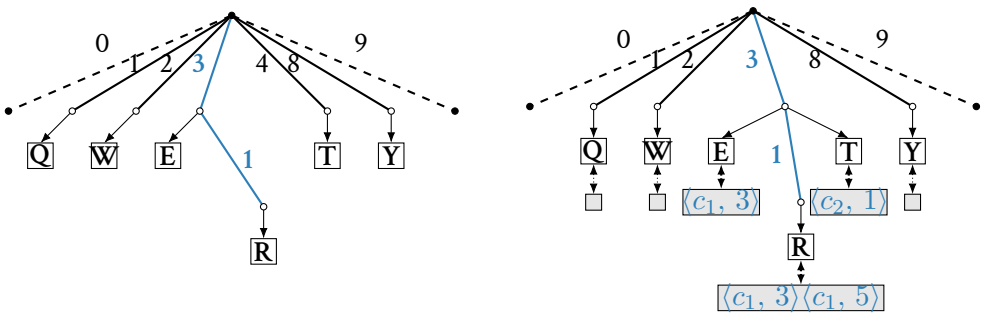
Une solution consiste à relocaliser les identifiants. En d'autres termes, les identifiants sont modifiés au profit d'identifiants plus courts et conservant l'ordre dense des éléments de la séquence. Toutefois, supprimer puis réinsérer l'ensemble des éléments dans un document neuf ne suffit pas : le trafic engendré est élevé et la concurrence est susceptible de cloner les éléments. Il faut donc parvenir à prendre une décision globale sur la relocalisation ce qui revient à obtenir un consensus en contexte réparti. Les consensus sont coûteux à obtenir dans les systèmes à large échelle où le nombre de répliques est grand, et particulièrement lorsque ces répliques ne sont pas accessibles en permanence.

Afin d'éviter les baisses de performance ainsi que les mécanismes de relocalisation, une autre solution consiste à obtenir des identifiants suffisamment courts pour qu'ils ne nécessitent pas d'être relocalisé. La section suivante s'attache à décrire les difficultés liées à l'allocation d'identifiants à taille variable.

2.2 Définition du problème

Nous nous intéressons aux approches les plus proches de notre spécification des séquences : celles générant des identifiants dont la taille est variable et définie à la génération.

Chemins. Les identifiants de taille variable peuvent être représentés comme la concaténation d'éléments basiques (e.g. des entiers). La séquence en résultant peut être représentée grâce à une structure d'arbre où les éléments de la séquence sont stockés sur les nœuds et où les arêtes sont étiquetées de telle sorte qu'un chemin de la racine jusqu'au nœud forme l'identifiant de l'élément. Par exemple, un caractère dont l'identifiant serait [3.1] serait accessible en suivant l'arête étiquetée 3, puis l'arête étiquetée 1. Plus formellement, la séquence est un arbre où chaque nœud peut contenir une valeur, i.e., un élément (dans l'alphabet $\mathcal{A}$) de la séquence. L'arbre est un ensemble de paires $\langle \mathcal{P} \subset \{N\}^*, \mathcal{A} \rangle$, i.e., chaque élément est associé à un chemin. De plus, un ordre dense et total $(\mathcal{P}, <_{\mathcal{P}})$ permet d'ordonner les chemins et de retrouver l'ordre des éléments de la séquence. Notation : un chemin composé de e arêtes étiquetées $\ell_1, \ell_2, \ldots, \ell_e$ est noté $[\ell_1.\ell_2 \ldots \ell_e]$.



(a) L'arbre représentant la séquence est construit grâce à l'union des identifiants et utilise principalement les chemins pour ordonner ses caractères.

(b) L'arbre utilise les désambiguateur afin de maintenir un état équivalent, même en présence d'insertions concurrentes ayant résulté en des chemins identiques. Par souci de simplicité, seuls les désambiguateurs des caractères E, R, et T sont affichés.

FIGURE 2.7 – Arbres contenant la séquence QWERTY.

La figure 2.7a montre l'arbre d'arité maximale 10 représentant la séquence. Un auteur insère les caractères QWTY les uns à la suite des autres. Les chemins en résultant sont [1], [2], [4] et [8] respectivement. L'insertion du caractère E entre les paires $\langle[2],\ \texttt{W}\rangle$ et $\langle[4],\ \texttt{T}\rangle$ engendre la paire suivante : $\langle[3],\ \texttt{E}\rangle$. Pour insérer le caractère R, l'arbre doit être agrandi afin d'accueillir le nouveau chemin entre E et T. Le chemin en résultant est [3.1]. Insérer un nouveau caractère entre E et R résulterait en une autre augmentation de la profondeur de l'arbre. Le chemin serait [3.0.X] où $0 < X < 10$. L'ordre total $(\mathcal{P},\ <_{\mathcal{P}})$ permet de retrouver la séquence QWERTY.

Désambiguation des cas concurrents. L'ordre des chemins $(\mathcal{P},\ <_{\mathcal{P}})$ est un ordre total lorsqu'un seul auteur écrit. Toutefois, cela devient un ordre partiel lorsque plusieurs auteurs sont impliqués dans l'édition. Par exemple, si deux auteurs insèrent un caractère au même endroit dans le document, au même moment, le même chemin pour les deux caractères risque d'être alloué. Dans ce cas, l'ordre des caractères n'est pas strictement défini et peut rompre la propriété de convergence des répliques. Les désambiguateurs utilisent des marqueurs unique de manière globale pour fournir un ordre total même en présence d'édition concurrente. Ces marqueurs sont généralement constitués d'un identifiant de site unique associé à une horloge de Lamport [56]. Chaque paire $\langle chemin,\ element\rangle$ est associée à un désambiguateur.

Soit l'ensemble des identifiants $\mathcal{I}$ composés tels que $\mathcal{I} : \mathcal{P} \times \mathcal{A} \times \mathcal{D}$. La composition de l'ordre partiel $(\mathcal{P},\ <_{\mathcal{P}})$ et l'ordre total des désambiguateurs $(\mathcal{D},\ <_{\mathcal{D}})$ permet d'ordonner les éléments de la séquence de manière identique quelle que soit la réplique.

La figure 2.7b montre l'arbre contenant 6 éléments mais seulement 5 chemins distincts.

Tout d'abord, le premier collaborateur c_1 insère `QW`. Ensuite, les collaborateurs c_1 et c_2 insèrent en concurrence les caractères `E` et `T`, respectivement. Dans les deux cas, le chemin en résultant est $[3]$. Pour résoudre cette ambiguïté, le désambiguateur $\langle c_1,\, 3\rangle$ est associé au caractère `E` ; le désambiguateur $\langle c_2,\, 1\rangle$ est associé au caractère `T`. Recouvrer l'ordre des éléments consiste simplement à comparer, à chaque niveau de l'arbre, le chemin du niveau, puis l'identifiant de la réplique, puis l'horloge. Dans cet exemple, le caractère `E` précède le caractère `T` car $c_1 < c_2$. Ensuite, le collaborateur c_1 insère `Y` à la fin de la séquence. Enfin, il insère `R` entre `E` et `T`. Puisque ces derniers ont un chemin identique, l'espace est insuffisant entre ces deux chemins pour insérer un nouveau caractère. La profondeur de l'arbre doit augmenter. La fonction d'allocation doit choisir un chemin $[3.X]$ tel que $0 < X < 10$. En copiant le désambiguateur de `E` au premier niveau, cela assure que le nouvel identifiant suivra celui du caractère `E` et précédera celui du caractère `T`. Il est important de noter que les collaborateurs ne choisissent pas les désambiguateurs. Ainsi, la séquence de cet exemple aurait pu être `QWTREY` auquel cas une correction aurait été nécessaire. De plus, la complexité spatiale de ces désambiguateurs est bornée par leur chemin respectif. Par conséquent, le reste de ce chapitre se concentre sur ces chemins.

Choisir le bon chemin. La composante la plus critique des approches pour les séquences où les identifiants sont de taille variable consiste à choisir les chemins. L'algorithme 1 montre les instructions générales de telles approches. Il divise les opérations – insertion et suppression – entre la partie locale et la partie distante du schéma de réplication optimiste. Le cœur de l'algorithme est situé dans la partie locale de l'opération d'insertion où le chemin et le désambiguateur doivent être créés. La fonction CONVERT2PATH se débarrasse de la partie désambiguateur des identifiants en argument pour n'en garder que les chemins – transformés si nécessaire. Lorsqu'il n'y a pas de marques d'opérations concurrentes, les chemins sont simplement retournés tels quels. Dans le cas contraire, cette fonction convertit les identifiants en chemins préservant l'ordre $(\mathcal{P}, <_{\mathcal{P}})$. Par exemple, dans la figure 2.7b, le résultat de l'appel à la fonction CONVERT2PATH avec les identifiants du caractère `E` et du caractère `T` est la paire $\langle [3.0],\, [3.9]\rangle$. La fonction ALLOCPATH alloue un nouveau chemin entre ces bornes. ALLOCDIS décore le chemin pour garantir que l'identifiant – en tant que composition d'un chemin, d'un élément, et d'un désambiguateur – est bien positionné entre les identifiants qui ont servi à le créer selon la relation d'ordre $(\mathcal{I},\, <_{\mathcal{I}})$.

La fonction ALLOCPATH choisit le chemin dans l'arbre entre deux autres chemins p et q tels que p précède q (i.e. $p <_{\mathcal{P}} q$). Le nouveau chemin n est positionné entre ces bornes (i.e. $p <_{\mathcal{P}} n <_{\mathcal{P}} q$). La fonction ALLOCPATH doit choisir parmi les plus petits chemins disponibles afin de conserver de bonnes performances.

Les figures 2.8a et 2.8b illustrent les difficultés rencontrées lors de l'allocation des chemins composant les identifiants. Dans les deux cas, la fonction d'allocation utilise la stratégie suivante : la branche la plus à gauche avec la plus petite profondeur possible. Dans les deux cas, la séquence finale est `QWERTY`. Toutefois, les lettres ne sont pas insérées dans un

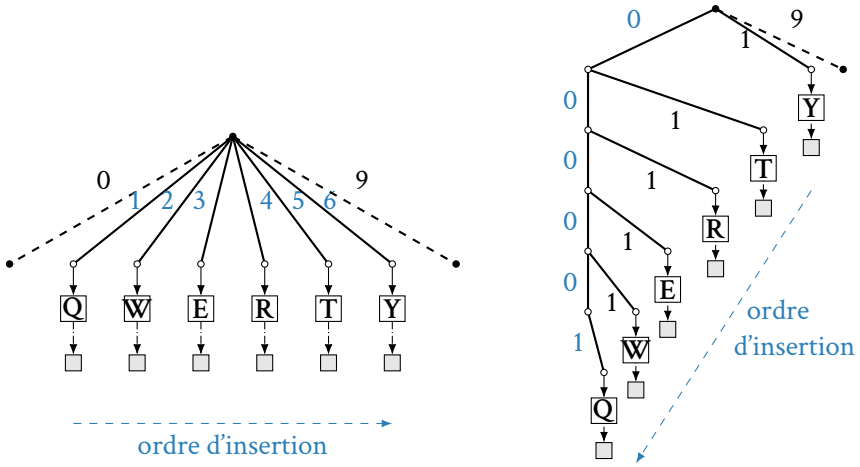
Algorithme 1 Séquences avec identifiants de taille variable.

```
1 : INITIALLY :
2 :     T ← ∅ ;                                          ▷ CRDT conçue pour les séquences
3 :
4 : LOCAL UPDATE :
5 :     on insert (previous ∈ I, α ∈ A, next ∈ I) :
6 :         let ⟨p, q⟩ ← convert2Path(previous, next) ;
7 :         let newPath ← allocPath(p, q) ;
8 :         let newDis ← allocDis(p, newPath, q) ;
9 :         broadcast('insert', ⟨newPath, α, newDis⟩) ;
10 :    on delete (i ∈ I) :
11 :        broadcast('delete', i) ;
12 :
13 : RECEIVED UPDATE :
14 :    on insert (i ∈ I) :                              ▷ une fois par identifiant
15 :        T ← T ∪ i ;
16 :    on delete (i ∈ I) :                              ▷ après l'exécution de insert(i)
17 :        T ← T \ i ;
```

ordre identique. Dans le premier cas, la lettre `Q` est insérée à l'indice 0, suivie de la lettre `W` à l'indice 1, suivie de la lettre `E` à l'indice 2, etc. Dans le second cas, la lettre `Y` est insérée à l'indice 0, suivie de `T` à l'indice 0 qui, par effet de bord, décale le `Y` à l'indice 1. La séquence en résultant est donc `TY`. L'insertion de `R` en position 0 décale toutes lettres de droite, etc.

Dans le premier cas (cf. figure 2.8a), l'opération d'insertion (`Q`, 0) est transformée en insert($\vdash$, `Q`, $\dashv$). Par conséquent, le chemin doit être alloué entre les bornes virtuelles [0] et [9]. Puisque la stratégie d'allocation consiste à réserver la branche la plus à gauche et de plus petite taille, le chemin résultant est [1]. Ensuite, l'opération (`W`, 1) est transformée en insert(i_Q, `W`, $\dashv$) résultant en le chemin [2], etc. Dans ce cas, la profondeur de l'arbre n'augmente jamais. À cet égard, la stratégie employée est très efficace. Cependant, elle n'est pas totalement optimale puisque l'arbre est prévu pour accueillir 8 identifiants alors que seulement 6 sont alloués.

Dans le second cas (cf. figure 2.8b), l'opération d'insertion (`Y`, 0) est transformée en insert($\vdash$, `Y`, $\dashv$) dont le résultat est le chemin [1]. Lors de l'insertion (`T`, 0), un chemin est requis entre la borne virtuelle du début de séquence [0] et la borne du caractère `Y` : [1]. Ainsi, la profondeur du chemin doit augmenter de telle sorte que le nouvel identifiant muni de l'ordre total $(\mathcal{I}, <_{\mathcal{I}})$ soit placé entre les deux. Considérons un ordre lexicographique, le chemin résultant est $[0.X]$ où X est choisi entre 0 et 10. Suivant la stratégie, le chemin est [0.1] pour la lettre `T`. Puis [0.0.1] pour la lettre `R`, etc. La taille des chemins alloués augmente très rapidement. La stratégie d'allocation s'avère inefficace dans ce cas.



(a) Allocation presque optimale (b) Allocation pire cas

FIGURE 2.8 – Une même stratégie d'allocation face à des comportements d'édition différents.

Cet exemple montre à quel point l'ordre d'insertion des éléments affecte la longueur des chemins alloués. Malheureusement, ni la séquence d'édition, ni sa taille ne sont connues par avance. Passer d'indices locaux mutables et optimaux d'une séquence classique à des identifiants non-mutables de taille variable d'une séquence répliquée a un coût qu'il convient d'analyser afin de proposer une fonction d'allocation efficace. Le problème scientifique que l'on souhaite résoudre est le suivant :

Définition du problème 1. *Soit la séquence d'identifiants* $s(I) = id_1.id_2 \ldots id_I$, *et* $s(I+1) = s(I) \cup \textsc{insert}(p, \_, n)$ *où* $p, q \in s(I)$ *et* $p <_{\mathcal{I}} q$. *Soit* $|s(I)|$ *la taille de la représentation binaire de la séquence. La fonction* INSERT *doit allouer des identifiants tels que :*

$$|s(I+1)| - |s(I)| < \mathcal{O}(I) \tag{2.3}$$

En d'autres termes, les chemins alloués par la fonction d'allocation ALLOCPATH, principal élément de la fonction INSERT, doivent garantir des chemins dont la taille est sous-linéaire par rapport au nombre d'insertions dans la séquence. Logoot et Treedoc échouent à fournir des identifiants de taille bornée de cette façon dans les cas d'insertions de gauche à droite et de droite à gauche. Ces deux approches nécessitent l'exécution périodique d'un mécanisme de relocalisation des identifiants afin de conserver de bonnes performances.

La section suivante présente LSEQ, une fonction d'allocation d'identifiants dont la taille croît de manière polylogarithmique par rapport au nombre d'insertions effectuées dans la séquence. Par conséquent, aucun mécanisme additionnel de relocalisation n'est nécessaire.

2.3 LSeq : une fonction d'allocation polylogarithmique

LSeq (*polyLogarithmic SEQuence*) est une fonction d'allocation d'identifiants de taille variable. L'algorithme 2 montre les instructions de LSeq : la fonction allocPath choisit le chemin associé à chaque élément afin d'encoder sa position relative à l'égard de ses éléments adjacents dans la séquence. Dans le but de conserver de bonnes performances, la profondeur de l'arbre doit rester aussi petite que possible.

Algorithme 2 Allocation des chemins selon LSeq.

```
1 : let boundary ← 10 ;                                   ▷ Une constante arbitraire
2 : let h : ℕ → (𝒫 × 𝒫 → 𝒫) ;                            ▷ Obtient la sous-fonction d'allocation

3 : function allocPath(p, q ∈ 𝒫) → 𝒫
4 :     let ⟨depth, _⟩ ← getDepthInterval(p, q) ;
5 :     return h(depth)(p, q) ;          ▷ Reporte l'appel à une sous-fonction d'allocation
6 : end function

7 : function left-to-right(p, q ∈ 𝒫) → 𝒫
8 :     let ⟨depth, interval⟩ ← getDepthInterval(p, q);     ▷ #1 niveau du nouveau chemin
9 :     let step ← min(boundary, interval) ;       ▷ #2 espace maximum entre deux chemins
10 :    return subPath(p, depth) + rand(0, step) ;          ▷ #3 crée le nouveau chemin
11 : end function

12 : function right-to-left(p, q ∈ 𝒫) → 𝒫
13 :    let ⟨depth, interval⟩ ← getDepthInterval(p, q);                          ▷ #1
14 :    let step ← min(boundary, interval) ;                                     ▷ #2
15 :    return subPath(q, depth) − rand(0, step) ;                               ▷ #3
16 : end function

17 : function getDepthInterval(p, q ∈ 𝒫) → ℕ × ℕ
     ▷ Où y a-t-il suffisamment d'espace pour 1 chemin ?
18 :    let depth ← 0 ; interval ← 0 ;
19 :    while (interval < 2) do
20 :        depth ← depth + 1 ;
21 :        interval ← subPath(q, depth) − subPath(p, depth) ;
22 :    end while
23 :    return ⟨depth, interval⟩ ;
24 : end function
```

La fonction allocPath calcule la distance entre les deux chemins adjacents. L'objectif est de trouver le plus petit niveau de l'arbre exponentiel (cf. §2.3.1) contenant assez d'espace pour accueillir le nouvel élément. Ensuite, une fonction de hachage h (cf. §2.3.3) reporte l'allocation du chemin à une sous-fonction d'allocation (cf. §2.3.2) selon la profondeur trou-

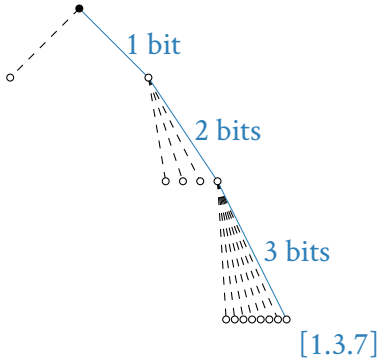


FIGURE 2.9 – Arbre exponentiel.

vée. La sous-fonction d'allocation calcule alors l'intervalle à disposition et la profondeur du nouveau chemin. Si l'intervalle est plus grand que la limite imposée, alors ce premier est restreint. Le nouveau chemin commence à partir de l'un des chemins adjacents, tronqué à la profondeur calculée, auquel est ajouté ou soustrait une valeur aléatoire dans le nouvel intervalle.

Ainsi, trois éléments principaux composent la fonction d'allocation des chemins de LSEQ :

(i) une structure d'arbre dont l'arité maximale augmente avec la profondeur ;

(ii) deux sous-fonctions d'allocation conçues pour gérer des comportements d'édition opposés ;

(iii) une fonction assignant à chaque profondeur de l'arbre une sous-fonction d'allocation parmi celles disponibles.

Individuellement, ces éléments n'arrivent pas à construire des chemins dont la taille est sous-linéaire par rapport au nombre d'insertions dans la séquence. En revanche, utilisés simultanément, ils permettent à LSEQ de résoudre ce problème.

Cette section détaille chacun de ces éléments avant de résumer le fonctionnement de LSEQ au travers d'un exemple.

2.3.1 Arbre exponentiel

Un arbre exponentiel est une structure d'arbre dont chacun des nœuds possède une arité maximale k fois supérieure à celle de son parent. Ainsi, la progression du nombre de fils dans une branche est exponentielle. Un chemin dans un tel arbre nécessite un bit additionnel par niveau de l'arbre.

L'intuition derrière l'utilisation de cette structure d'arbre est la suivante : une augmentation de la profondeur de l'arbre signifie que le nombre d'insertions dans la séquence est suffisant pour nécessiter un plus large champ d'identifiants. Ainsi, au lieu d'ouvrir un espace aussi large qu'auparavant, l'espace est encore élargi afin que la séquence puisse tirer profit de ce champ d'identifiants plus longuement.

La figure 2.9 montre un arbre exponentiel. L'arbre commence avec une arité maximale de 2. Puis chacun des fils a une arité maximale de 4, ensuite une arité maximale de 8, etc. Un chemin de profondeur 3 tel que [1.3.7] nécessite alors 1+2+3 = 6 bits pour sa représentation en mémoire. Plus généralement, un chemin de profondeur e requiert $\sum_{i=1}^{e} i = \frac{e^2+e}{2}$ bits. Cette progression quadratique en fonction de la profondeur rend essentielle la prise en charge des comportements d'édition conduisant au pire cas (cf. figure 2.8b). En particulier, un comportement aussi simple que l'édition de droite à gauche doit impérativement être géré.

2.3.2 Sous-fonctions d'allocation

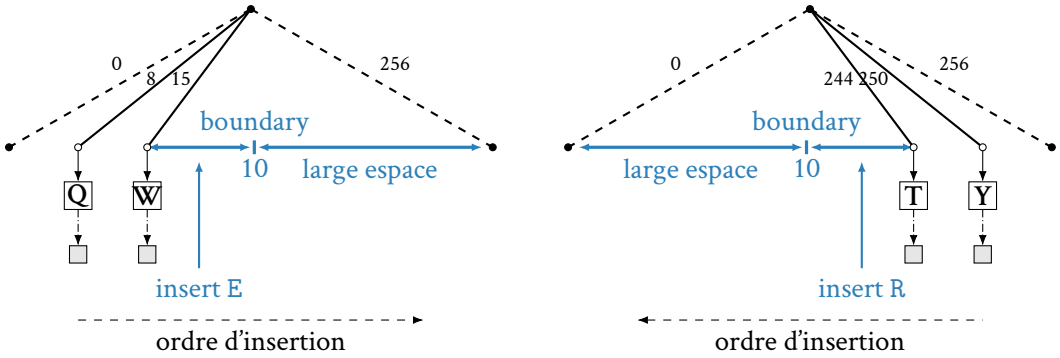
L'allocation d'identifiants optimale suppose de connaître le nombre et la position des éditions successives. Avec cette connaissance, les identifiants auraient une croissance logarithmique par rapport à la taille document. Malheureusement, aucune de ces deux informations n'est disponible dans l'édition collaborative en temps réel.

Dans ces conditions, nous supposerons que le comportement d'édition est une suite d'éditions successives – comme l'édition de gauche à droite et l'édition de droite à gauche – ou à des positions aléatoires, ou enfin, une composition de celles-ci. Afin de gérer ces types d'édition, employer une unique fonction d'allocation conçue pour l'édition de gauche à droite ne suffit pas.

La figure 2.10 décrit le fonctionnement des deux sous-fonctions d'allocation employées par LSeq. Les arbres possèdent une arité maximale de 256 fils à ce niveau. Lorsqu'une insertion est effectuée, un nombre aléatoire est tiré dans l'espace disponible avec une borne supérieure imposée (la *boundary*). Cette borne permet de laisser un petit espace libre entre les identifiants des caractères. Comme le montre la figure 2.10a, si la borne commence proche du caractère précédent, alors la fonction d'allocation est adaptée à l'édition de gauche à droite car elle laisse un espace important pour les insertions à venir à droite du nouvel élément. Au contraire, comme le montre la figure 2.10b, si la borne commence proche du caractère suivant, alors la fonction d'allocation est adaptée à l'édition de droite à gauche.

2.3.3 Choix de sous-fonction d'allocation

L'utilisation de plusieurs sous-fonctions d'allocation oblige à choisir parmi celles-ci. Cela pose deux problèmes, à savoir, (i) quel événement déclenche une prise de décision, et lorsque cela survient, (ii) quelle sous-fonction employer.



(a) Sous-fonction d'allocation conçue pour l'édition de gauche à droite

(b) Sous-fonction d'allocation conçue pour l'édition de droite à gauche

FIGURE 2.10 – Deux sous-fonctions d'allocation.

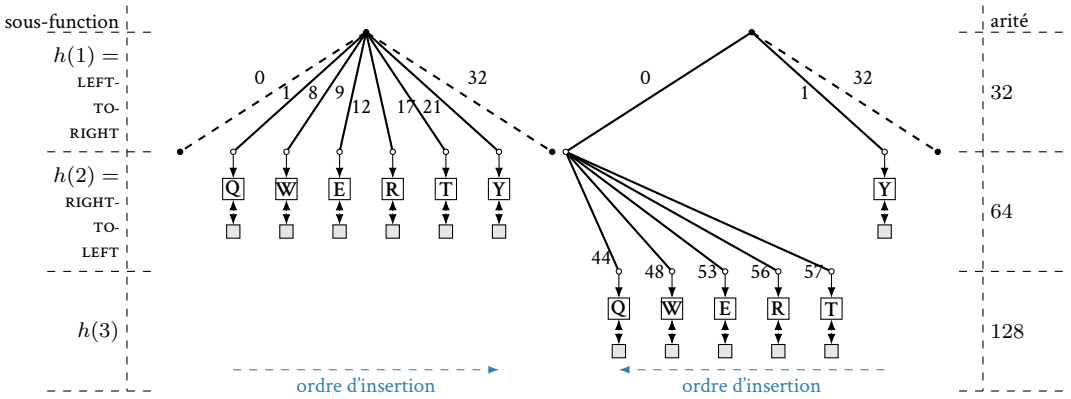
Pour chaque niveau de l'arbre, LSEQ assigne une sous-fonction d'allocation grâce à une fonction de hachage. Cette fonction doit retourner les mêmes résultats quel que soit le participant. Pour cela, un germe, commune à tous les participants, est placé dans la séquence répliquée. Lorsqu'un nouveau participant arrive, il réplique le germe avec la séquence.

Afin de ne privilégier aucun comportement d'édition, le choix parmi les sous-fonctions doit être uniformément réparti. De plus, afin de ne pas faciliter les comportements malveillants, la fonction de hachage doit rendre les choix imprévisibles.

2.3.4 Conclusion

L'idée générale de LSEQ consiste à accepter la perte de quelques niveaux des chemins composants ses identifiants pour peu que les futures allocations compensent ces pertes.

La figure 2.11 montre le résultat de la stratégie d'allocation LSEQ après deux comportements d'édition : l'un écrit de gauche à droite, l'autre écrit de droite à gauche. Dans les deux cas, la séquence est `QWERTY` ; l'arité maximale commence à 2^5 et est doublée à chaque niveau ; la fonction de hachage désigne les sous-fonctions conçues pour l'édition de gauche à droite, et pour l'édition de droite à gauche, pour les niveaux 1 et 2 respectivement. La figure 2.11a montre le cas de l'édition de gauche à droite. Dans ce cas, le comportement d'édition attendu par la sous-fonction d'allocation est bon, et les chemins restent alloués au premier niveau de l'arbre. D'un autre côté, la figure 2.11b montre le cas de l'édition de droite à gauche. Dans ce cas, la sous-fonction d'allocation du premier niveau n'est pas adaptée. La profondeur de l'arbre augmente dès la seconde insertion. En revanche, puisque la sous-fonction d'allocation choisie à ce niveau est adaptée, la profondeur de l'arbre n'augmente pas d'avantage. Si le comportement d'édition reste ainsi, l'allocation finira par compenser



(a) Édition de gauche à droite dans LSeq (b) Édition de droite à gauche dans LSeq

FIGURE 2.11 – Exemple d'arbres exponentiels LSeq remplis par deux comportements d'édition antagonistes résultant en une même séquence `QWERTY`.

les pertes subies par le mauvais choix de sous-fonction au premier niveau de l'arbre.

La section suivante présente l'analyse en complexité de LSeq. En particulier, elle s'attache à montrer la complexité spatiale sous-linéaire des identifiants ainsi que les conditions sous lesquelles elle s'applique.

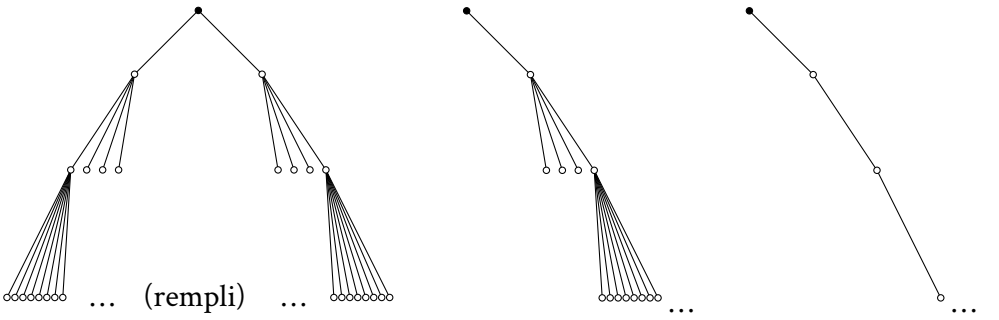
2.4 Analyse en complexité

Dans l'édition de texte, la plupart des comportements d'édition peuvent empiriquement être résumés à la composition de deux comportements basiques.

Le comportement d'édition aléatoire. L'auteur insère de nouveaux éléments à ce qui semble être des positions aléatoires dans la séquence. Par exemple, ce comportement peut apparaître lors de correction orthographique, e.g. l'auteur écrit `QWETY` et réalise qu'il lui manque le `R`. Il l'ajoute donc dans un second temps.

Le comportement d'édition monotone L'auteur insère de nouveaux éléments les uns à la suite des autres (avant ou après l'élément le plus récemment inséré). Par exemple, lorsqu'un auteur écrit `QWERTY`, il commence généralement par le caractère `Q` jusqu'à arriver au caractère `Y`. À l'opposé, les insertions dans un historique se font en tête pour des raisons pratiques. Ces comportements sont respectivement notés l'édition de droite à gauche, et l'édition de gauche à droite.

Cette section concentre l'analyse en complexité de LSeq sur les comportements d'édition susmentionnés auxquels est ajoutée une analyse en pire cas. Il est important de noter



(a) Arbre LSeq rempli par un comportement d'édition aléatoire.

(b) Arbre LSeq rempli par un comportement d'édition monotone.

(c) Arbre LSeq rempli dans le pire cas.

FIGURE 2.12 – Influence du comportement d'édition sur l'arbre exponentiel.

que le comportement d'édition monotone constitue un cas défavorable puisqu'il tend à déséquilibrer l'arbre stockant la séquence répliquée. Aussitôt que les participants commencent à éditer à différents endroits – ce qui est plus proche d'un cas d'utilisation réel – l'arbre commence à s'équilibrer. L'analyse ne fait pas mention de cas moyen puisqu'il nécessite de connaître la distribution des positions des insertions effectuées par des être humains ce qui représente une tâche complexe. Enfin, l'analyse en complexité se concentre sur la représentation de la séquence sous forme d'arbre. Cette représentation, par rapport à une liste, possède l'avantage d'une structure de séquence plus compacte au prix d'accès plus lents. D'autres structures existent proposant d'autres compromis. Cependant, il est important de noter que la complexité spatiale d'un identifiant demeure inchangée.

2.4.1 Complexité spatiale

Dans cette analyse, la complexité spatiale d'un seul identifiant est différenciée de la complexité spatiale de l'arbre stockant la séquence. La première est importante puisqu'elle influence directement la complexité en communication. En effet, chaque identifiant est propagé à tous les possesseurs de répliques. La seconde est importante puisqu'elle représente le mémoire allouée localement par chaque éditeur collaboratif.

La section 2.3 spécifie que chaque nouvelle concaténation dans le chemin de l'identifiant requiert un bit additionnel pour l'encoder. Cela représente au total $\mathcal{O}(e^2)$ bits pour encoder le chemin, où e est la profondeur du chemin dans l'arbre. Heureusement, cette profondeur est bornée selon le comportement d'édition remplissant l'arbre exponentiel.

Comportement d'édition aléatoire. L'édition aléatoire remplit l'arbre à des positions aléatoires. En conséquence, l'arbre reste équilibré (cf. figure 2.12a). Étant exponentiel, l'arbre

TABLE 2.1 – Bornes supérieures de la complexité spatiale de LSEQ, Logoot, et Treedoc. I est le nombre d'insertions effectuées dans la séquence.

COMPORTEMENT D'ÉDITION	ESPACE DE LSEQ		ESPACE DE LOGOOT / TREEDOC	
	IDENTIFIANTS	SÉQUENCE	IDENTIFIANTS	SÉQUENCE
Édition aléatoire	$\mathcal{O}(\log I)$	$\mathcal{O}(I \log I)$	$\mathcal{O}(\log I)$	$\mathcal{O}(I)$
Édition monotone	$\mathcal{O}((\log I)^2)$	$\mathcal{O}(I \log I)$	$\mathcal{O}(I)$	$\mathcal{O}(I)$
Pire cas	$\mathcal{O}(I^2)$	$\mathcal{O}(I^2)$	$\mathcal{O}(I)$	$\mathcal{O}(I)$

peut stocker $\sum_{i=1}^{k} 2^{(i^2-i)/2}$ éléments, où k est la profondeur de l'arbre. Ainsi, la borne supérieure sur la profondeur des chemins est $\mathcal{O}(\sqrt{\log I})$ concaténations, où I est le nombre d'insertions. Puisque les chemins nécessitent $\mathcal{O}(e^2)$ bits pour être encodés, les chemins ont une complexité spatiale optimale de $\mathcal{O}(\log I)$. Ce résultat s'applique à toutes les approches utilisant des identifiants de taille variable [84, 105]. Toutefois, puisque LSEQ utilise une structure d'arbre factorisant les parties communes des identifiants, la complexité spatiale totale n'est pas la somme des identifiants, mais seulement $\mathcal{O}(I \log I)$.

Comportement d'édition monotone. L'édition monotone remplit seulement une branche de l'arbre (cf. figure 2.12b). Pourtant, comme l'arité maximale d'un nœud augmente lorsque sa profondeur augmente, la croissance de l'arbre ralentit au cours des insertions. Dans ce cas, l'arbre stocke jusqu'à $2^{e+1} - 1$ éléments, où e est la profondeur de l'arbre. Ainsi, le nombre de concaténations composant le chemin est $\mathcal{O}(\log I)$, où I est le nombre d'insertions. La représentation binaire des chemins grandissant de façon quadratique, la complexité spatiale des identifiants est $\mathcal{O}((\log I)^2)$. Globalement, la complexité spatiale de l'arbre reste la même, à savoir $\mathcal{O}(I \log I)$.

Pire cas. Le pire cas s'obtient par l'insertion systématique d'éléments à l'endroit où l'espace d'allocation est le plus faible. Par exemple, lorsque la sous-fonction d'allocation employée est conçue pour l'édition de gauche à droite, peu d'espace est laissé disponible à gauche de ce niveau. Dans le pire cas, chaque insertion augmente la profondeur de l'arbre (cf. figure 2.12c). Ainsi, après I insertions, l'arbre comprend e éléments, où e est la profondeur de l'arbre. Chaque nouveau chemin possède tous les autres chemins. Ainsi, la complexité spatiale des identifiants est $\mathcal{O}(I^2)$ et la complexité spatiale de l'arbre est elle aussi $\mathcal{O}(I^2)$.

Résumé. Le tableau 2.1 résume les complexités spatiales de LSEQ. En particulier, la croissance des identifiants est attendue entre un logarithme et un polylogarithme. La croissance est donc bien sous-linéaire par rapport au nombre d'insertions effectuées dans la séquence. Pour atteindre cette amélioration, LSEQ sacrifie son pire cas en le rendant quadratique. Toutefois, ce pire cas est rendu difficile à obtenir. En effet, les deux sous-fonctions d'allocation aux buts antagonistes résolvent leurs limitations respectives. De plus, si un utilisateur

TABLE 2.2 – Bornes supérieures de la complexité temporelle de LSEQ. I est le nombre d'insertions effectuées dans la séquence.

COMPORTEMENT D'ÉDITION	TEMPS		
	LOCAL		DISTANT
	INS	DEL	INS / DEL
Édition aléatoire	$\mathcal{O}(\sqrt{\log I})$	$\mathcal{O}(1)$	$\mathcal{O}(\log I + \sqrt{\log I})$
Édition monotone	$\mathcal{O}(\log I)$	$\mathcal{O}(1)$	$\mathcal{O}((\log I)^2 + \log I)$
Pire cas	$\mathcal{O}(I)$	$\mathcal{O}(1)$	$\mathcal{O}(I)$

malintentionné tente de produire ce pire cas, la différence entre la croissance attendue et la croissance observée serait telle que le coupable serait aisément identifiable. Des mesures pourraient alors être prises. Le tableau 2.1 montre aussi que, comparé à l'état de l'art, LSEQ améliore grandement la taille des identifiants mais voit sa taille globale légèrement dégradée, i.e., de $\mathcal{O}(I)$ à $\mathcal{O}(I \log I)$. Le compromis reste avantageux car les communications héritent de l'amélioration sur les identifiants.

2.4.2 Complexité temporelle

L'analyse en complexité temporelle fournit des indices sur l'évolution des performances de chaque opération au fur et à mesure des insertions. Ces opérations sont découpées entre leur exécution locale et leur exécution distante. Tout comme pour la complexité spatiale, l'analyse se porte sur les trois comportements d'édition suivant : aléatoire, monotone et pire cas.

Insertion locale. Cette opération consiste simplement à construire un nouveau chemin en fonction des chemins adjacents. Par conséquent, la complexité dépend de la taille de ces chemins qui grandissent en fonction du comportement d'édition. Les comportements d'édition aléatoire, monotone et pire cas conduisent respectivement à une croissance de l'arbre en $\mathcal{O}(\sqrt{\log I})$, $\mathcal{O}(\log I)$ et $\mathcal{O}(I)$. La complexité temporelle de la partie locale de l'insertion suit donc ces mêmes complexités.

Suppression locale. L'opération consiste à propager l'identifiant à supprimer à tous les participants. L'opération se fait donc en temps constant $\mathcal{O}(1)$ quel que soit le comportement d'édition.

Insertion et suppression distante. Les opérations d'insertion et de suppression locales retournent toutes les deux des identifiants que la partie distante de l'opération correspondante est en charge d'intégrer. Les instructions composant ces opérations distantes sont identiques : il s'agit principalement d'une exploration de l'arbre. Par conséquent, leur com-

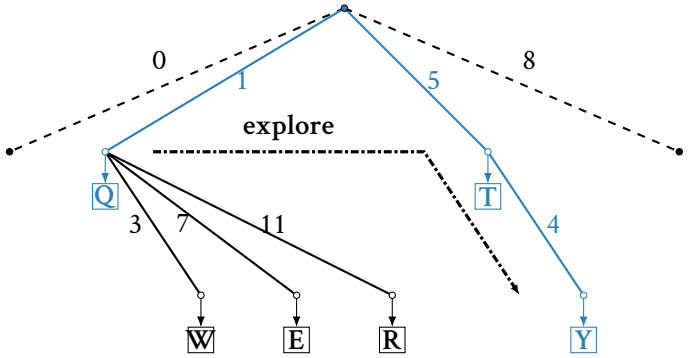


FIGURE 2.13 – Exemple de parcours effectué lors de la recherche de l'identifiant en position 5 correspondant au caractère Y.

plexité temporelle est identique. Encore une fois, les comportements d'édition aléatoire, monotone et pire cas conduisent respectivement à une croissance de l'arbre exponentiel de l'ordre de $\mathcal{O}(\sqrt{\log I})$, $\mathcal{O}(\log I)$, et $\mathcal{O}(I)$. Les fils de chaque nœud de l'arbre sont ordonnés selon leur chemin. Ainsi, une recherche dichotomique récursive permet d'atteindre la feuille de l'arbre recherchée. La suppression distante supprime les nœuds qu'elle traverse si ce nœud ne possède qu'un seul élément parmi ses sous-arbres. La complexité temporelle de la recherche dichotomique dépend du niveau l à laquelle elle est effectuée : $\mathcal{O}(\log 2^l)$. Les recherches à répétition conduisent à une borne supérieure en $\mathcal{O}(\sum_{i=1}^{e}(\log 2^i))$ où e est la profondeur de l'arbre. En remplaçant la variable e, les bornes supérieures sur la complexité temporelle des opérations deviennent $\mathcal{O}(\log I + \sqrt{\log I})$ pour le comportement aléatoire, $\mathcal{O}((\log I)^2 + \log I)$ pour le comportement monotone et $\mathcal{O}(I)$ dans le pire cas. Pour ce dernier, cela signifie qu'une comparaison par élément est nécessaire afin d'atteindre la feuille la plus profonde.

Interface arbre – séquence. La structure de données répliquée est un arbre mais l'objet manipulé par l'éditeur collaboratif est une séquence. Il est donc nécessaire de fournir les fonctionnalités d'accès élémentaires afin de faire le lien entre l'arbre et la séquence. Une fonction additionnelle de LOOKUP est requise. Son but est de récupérer l'élément à l'indice spécifié dans la séquence, et inversement. Puisque la structure sous-jacente est un arbre, l'accès n'est pas direct.

Chaque nœud de l'arbre possède un compteur désignant la quantité d'éléments présents dans ses sous-arbres. Chaque opération d'insertion ou de suppression met à jour ces compteurs lorsqu'elle explore le chemin. Ensuite, calculer l'indice d'un identifiant revient à compter le nombre d'éléments chez les voisins des nœuds explorés, en commençant par l'élément de l'une des extrémités de l'arbre. Hélas, selon le comportement d'édition, les performances de cette fonction peuvent se dégrader très rapidement.

La figure 2.13 montre un exemple de recherche d'un identifiant dans un arbre représentant

TABLE 2.3 – Bornes supérieures de la complexité temporelle de la fonction LOOKUP de LSEQ. I est le nombre d'insertions effectuées dans la séquence.

COMPORTEMENT D'ÉDITION	TEMPS LOOKUP
Édition aléatoire	$\mathcal{O}(2^{\sqrt{\log I}})$
Édition monotone / pire cas	$\mathcal{O}(I)$

la séquence `QWERTY`. La recherche concerne la position 5 correspondant au caractère `Y`. Ici, l'arbre est parcouru en partant du fils gauche de la racine. Ce fils sait que ses sous-branches contiennent 3 caractères. L'algorithme en déduit que le prochain fils de la racine est à la position 4. En examinant ce second fils, l'algorithme déduit que l'identifiant recherché est inclus dans l'un de ses fils. Ainsi, il continue l'exploration dans ce sous-arbre. Lorsqu'il examine le nœud contenant `Y`, il sait que celui-ci est en position 5. Par conséquent, il retourne le chemin qu'il a emprunté pour y parvenir, à savoir [5.4]. Nous pouvons observer dans cet exemple que si la recherche était partie de la droite, elle n'aurait eu à parcourir qu'un seul élément intermédiaire (contenant le caractère `T`), et aurait donc été plus efficace.

Le comportement d'édition aléatoire conduit à un arbre dont la profondeur est bornée par $\mathcal{O}(\sqrt{\log I})$. Dans pareil cas, une très faible portion de l'arbre est explorée. Pour obtenir la borne supérieure, il faut que le nœud à rechercher se trouve en plein milieu de la séquence, ce qui force à examiner la moitié des voisins du nœud exploré à chaque niveau de l'arbre. Puisque chaque nœud peut posséder jusqu'à deux fois plus de fils que son parent, le nombre de voisins à examiner double également. Au total, la somme de ces voisins est égale au nombre de nœuds présents dans une branche du plus profond niveau, c'est-à-dire $\mathcal{O}(2^{\sqrt{\log I}})$ éléments.

Grâce à un raisonnement identique, la fonction de LOOKUP après un comportement d'édition monotone – ce qui constitue le pire cas – est bornée par $\mathcal{O}(I)$. En effet, seulement une branche est remplie et explorée. Ici, le compteur maintenu par chaque nœud devient inutile puisque les nœuds ne possèdent qu'un seul et unique élément. L'exploration s'achève au plus profond des niveaux contenant $\mathcal{O}(2^{\log_2 I})$ éléments dont la moitié doit être examinée. La complexité temporelle dans ce cas est : $\mathcal{O}(I)$.

Résumé. Les tableaux 2.2 et 2.3 synthétisent la complexité temporelle des opérations. L'arbre exponentiel permet d'obtenir des opérations de mise-à-jour de l'état efficace. En revanche, l'opération LOOKUP qui permet d'interfacer l'arbre et la séquence souffre de ce choix de structure. Entre autres, le comportement d'édition monotone constitue le pire cas pour cette opération où l'accès se fait en temps linéaire. Pour comparaison, une liste offre des accès en temps constant et retrouve la position d'insertion à partir d'un identifiant en

temps logarithmique par rapport à la taille de la séquence. Heureusement, (i) la vue n'a pas besoin d'être mise-à-jour si l'indice du nouvel élément se trouve hors du champs de visibilité de l'utilisateur. Ainsi, la fonction LOOKUP peut s'arrêter plus tôt ; (ii) un accès rapide aux identifiants alloués les plus récents ainsi que les chemins qui lui sont adjacents efface le coût du LOOKUP puisque ces accès rapides peuvent être maintenus à jour pendant les insertions sans coût additionnel.

2.4.3 Conclusion

L'analyse en complexité révèle les améliorations apportées par LSEQ par rapport à l'état de l'art ainsi que leur coût. Dans le pire cas, la complexité de LSEQ est moins bonne que celle de l'état de l'art [84, 105]. D'un autre côté, cette concession permet d'améliorer ses performances sur les autres comportements d'édition considérés plus probables dans l'édition collaborative. Le résultat le plus important concerne la borne supérieure sur la complexité spatiale des identifiants, à savoir une croissance polylogarithmique par rapport au nombre d'insertions dans la séquence. L'importance de ce résultat provient du fait qu'il influence la complexité en communication de l'application qui l'utilise. En comparaison, les approches de l'état de l'art fournissent des identifiants dont la taille croît linéairement par rapport au nombre d'insertions dans la séquence. En outre, la complexité spatiale de la structure ainsi que la complexité temporelle de ses opérations montre que le choix d'un arbre pour représenter la séquence est efficace. Néanmoins, selon les préférences, le choix d'un vecteur – en tant que version aplatie de l'arbre – est possible afin d'améliorer les performances des opérations au prix d'un usage mémoire plus important [105]. La section suivante s'attache à valider ces analyses en complexité au travers d'expérimentations.

2.5 Validation

LSEQ est une fonction d'allocation d'identifiants pour les structures de données sans conflits conçues pour les séquences. Cette section a trois objectifs. Tout d'abord, elle examine la contribution de chacun des composants internes à LSEQ décrits en §2.3. Ensuite, elle valide expérimentalement les analyses en complexité de LSEQ faites en §2.4. Enfin, elle montre l'influence de la concurrence sur l'allocation d'identifiants.

2.5.1 Référence

Objectif : Fournir une référence aux futures expérimentations afin de montrer les améliorations et les dégradations de chaque composant.

Description : La simulation concerne trois documents artificiels construits respectivement par un comportement d'édition aléatoire, un comportement d'édition monotone de gauche à droite et un comportement d'édition monotone de droite à gauche. La fonction d'allocation utilise un arbre d'arité maximale constante et une seule sous-fonction d'alloca-

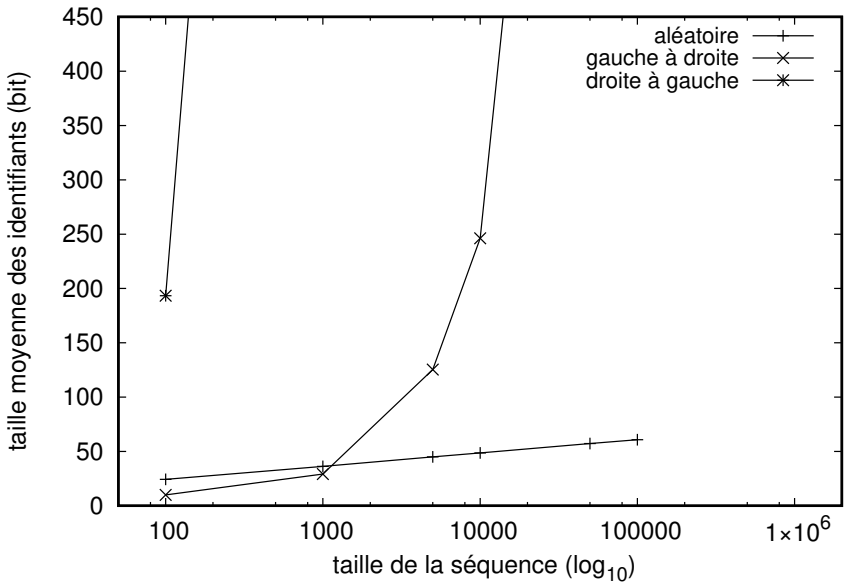


FIGURE 2.14 – Arbre à arité constante avec stratégie d'allocation adaptée à l'édition de gauche à droite. L'axe des abscisses montre la taille du document sur une échelle logarithmique en base décimale. L'axe des ordonnées montre la taille moyenne des chemins alloués.

tion conçue pour l'édition monotone de gauche à droite. Cette configuration correspond aux stratégies de l'état de l'art [84, 105]. La moyenne de la taille des identifiants est mesurée à 100, 1k, 5k, 10k, 50k, 100k insertions.

Résultat : La figure 2.14 montre les résultats de cette expérimentation. Nous observons tout d'abord que la taille des identifiants croît logarithmiquement dans le cas des éditions faites à des positions aléatoires. Ensuite, les comportements d'édition monotone conduisent tout deux à une croissance linéaire. Toutefois, l'édition de gauche à droite reste bien plus efficace que l'édition de droite à gauche.

Explication : Le comportement d'édition aléatoire conduit à une structure d'arbre équilibrée. Les identifiants alloués restent petits. Le comportement d'édition monotone de gauche à droite conduit à une croissance linéaire mais relativement faible. En effet, la stratégie d'allocation employée est conçue pour ce comportement : des branches sont laissées disponibles en vue des prochaines insertions. La taille des identifiants grandit donc lentement. Toutefois, comme l'arité maximale de l'arbre reste constante, la structure ne s'adapte pas à la taille du document, d'où la croissance linéaire. Dans le cas du comportement d'édition monotone de droite à gauche, l'augmentation de la taille des identifiants est très forte car la stratégie employée favorise le comportement d'édition opposé. Ainsi les insertions ont tôt fait de ne plus avoir d'espace disponible conduisant à la création d'un nouvel espace, i.e. la profondeur de l'arbre augmente. L'augmentation reste linéaire.

2.5.2 Arbre exponentiel

Objectif : Montrer que l'utilisation d'un arbre exponentiel permet d'allouer des chemins dont la taille croît de manière polylogarithmique par rapport au nombre d'insertions. Montrer que lorsque le comportement d'édition va à l'encontre de celui prévu, les identifiants alloués ont une croissance quadratique.

Description : Trois documents sont créés artificiellement par insertions successives de caractères. Pour chacun, un comportement d'édition différent est simulé : (i) aléatoire, (ii) monotone de gauche à droite et (iii) monotone de droite à gauche. La fonction d'allocation est conçue pour l'édition de gauche à droite et utilise un arbre exponentiel. La moyenne de la taille des identifiants est mesurée à 100, 1k, 5k, 10k, 50k, 100k, 500k insertions.

Résultat : La figure 2.15 montre les résultats des mesures effectuées pendant la simulation. Nous observons qu'avec un comportement d'édition aléatoire, et similairement aux mesures de références (cf. figure 2.14), la croissance des identifiants est optimale puisqu'elle suit un logarithme par rapport au nombre d'insertions dans le document. Lors de l'édition monotone de gauche à droite, la croissance des chemins est polylogarithmique par rapport au nombre d'insertions. En revanche, l'édition de droite à gauche entraîne une augmentation quadratique de la taille des identifiants.

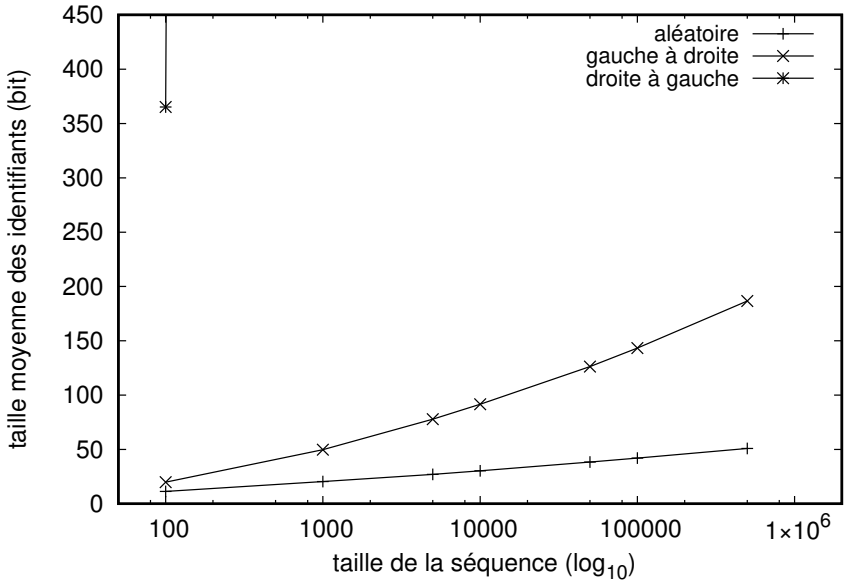


FIGURE 2.15 – Arbre exponentiel avec stratégie d'allocation adaptée à l'édition de gauche à droite. L'axe des abscisses montre la taille du document sur une échelle logarithmique en base décimale. L'axe des ordonnées montre la taille moyenne des chemins alloués.

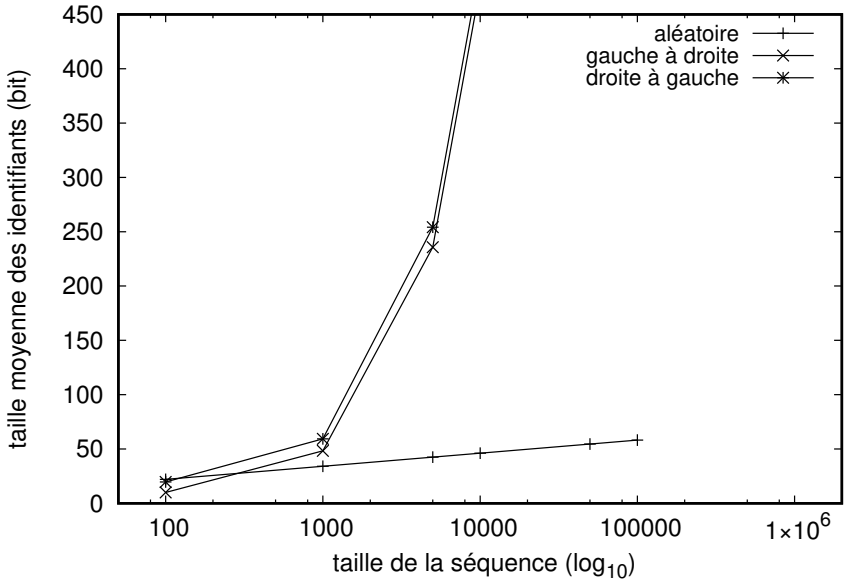


FIGURE 2.16 – Arbre à arité constante et deux sous-stratégies d'allocation. L'axe des abscisses montre la taille du document sur une échelle logarithmique en base décimale. L'axe des ordonnées montre la taille moyenne des chemins alloués.

Explication : L'édition à des positions aléatoires place les éléments au hasard dans la séquence. L'arbre est équilibré car nulle branche n'est favorisée. À terme, les branches proches de la racine sont remplies entièrement. Dans ce cas, les chemins sont de taille logarithmique ce qui constitue la borne minimale. Lors du comportement d'édition monotone de gauche à droite, les chemins les plus à gauche sont alloués réservant de l'espace aux insertions futures. De plus, puisque l'arité de l'arbre double à chaque niveau, il peut accueillir deux fois plus de chemins à un prix minime (1 bit additionnel par niveau). Pour cette même raison, la croissance des identifiants devient quadratique lors de l'édition de droite à gauche : quelques insertions suffisent à faire augmenter la profondeur de l'arbre. L'arité maximale augmente alors rapidement et le prix des identifiants explose. Dans ce cas, la taille des chemins croît de manière quadratique.

2.5.3 Sous-fonctions d'allocation

Objectif : Montrer que deux sous-fonctions d'allocation conçues avec des objectifs antagonistes et assignées à chaque profondeur de l'arbre permettent de gèrer les comportements d'édition simples : monotones et aléatoire. Montrer que la progression des identifiants reste linéaire par rapport à la taille du document.

Description : La simulation concerne trois documents grossissant au rythme des opérations d'insertion effectuées respectivement de gauche à droite, de droite à gauche et aléatoirement. La fonction d'allocation utilise un arbre dont l'arité maximale est constante. De plus, chaque niveau se voit attribuer une sous-fonction d'allocation, i.e., les niveaux pairs avec une fonction conçue pour l'édition de gauche à droite et les niveaux impairs avec une fonction adaptée à l'édition de droite à gauche. Nous mesurons la moyenne de la taille des chemins composant les identifiants à 100, 1k, 5k, 10k, 50k, 100k insertions.

Résultat : La figure 2.16 montre les résultats obtenus à la suite de ces simulations. Nous observons tout d'abord que les chemins restent logarithmiques sous un comportement d'édition aléatoire. Nous observons aussi que, pour les deux types d'édition monotone, la croissance est linéaire. De plus, les mesures sont quasiment identiques dans les deux cas.

Explication : Tout comme pour les expériences précédentes (cf. figure 2.14 et 2.15), l'édition à des positions aléatoires à pour effet d'équilibrer l'arbre représentant le document répliqué. Les chemins résultant de ce comportement grandissent de manière logarithmique. Grâce à une alternance des sous-fonctions d'allocation, la taille des chemins alloués lors des comportements d'édition monotone augmente lentement. Malgré tout, la progression reste linéaire. De plus, elle augmente deux fois plus rapidement que sans cette alternance avec un comportement d'édition favorable (cf. figure 2.14). En effet, dans le cas de ces simulations, un niveau sur deux est perdu car consommé trop vite : la sous-fonction d'allocation assignée à ce niveau n'était pas celle conçue pour le comportement d'édition courant.

2.5.4 Arbre exponentiel et sous-fonctions

Objectif : Montrer que l'utilisation combinée d'un arbre exponentiel et de sous-fonctions d'allocation permet de pallier leurs faiblesses respectives et d'obtenir une stratégie d'allocation sous-linéaire par rapport au nombre d'insertions effectuées dans le document.

Description : Trois documents chacun édité d'une façon différente : l'auteur du premier possède un comportement d'édition aléatoire ; l'auteur du second possède un comportement d'édition monotone de gauche à droite ; l'auteur du troisième de droite à gauche. La fonction d'allocation utilise un arbre exponentiel et deux sous-fonctions d'allocation conçues pour gérer les éditions monotones. Les sous-fonctions d'allocation sont assignées aléatoirement à chaque niveau de l'arbre. Les mesures sont faites à 100, 1k, 5k, 10k, 50k, 100k, 500k, 1M insertions.

Résultat : La figure 2.17 affiche les résultats obtenus lors de cette expérimentation. Nous observons que la progression de la taille des identifiants lors du comportement d'édition aléatoire reste logarithmique. Lors des deux autres comportements d'édition monotone, la croissance des identifiants suit une progression polylogarithmique par rapport au nombre d'insertions effectuées dans la séquence. Cette observation confirme l'analyse en complexité

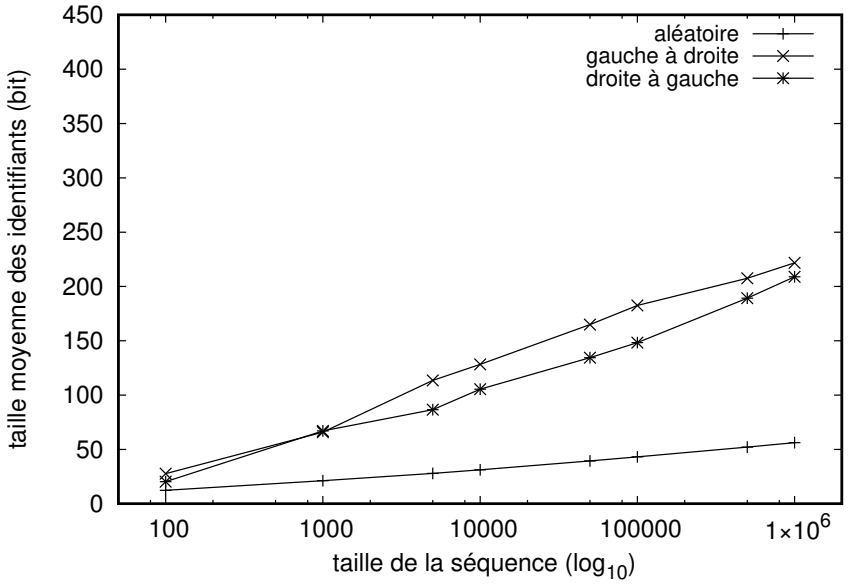


FIGURE 2.17 – Arbre exponentiel avec deux sous-stratégies d'allocation. L'axe des abscisses montre la taille du document sur une échelle logarithmique en base décimale. L'axe des ordonnées montre la taille moyenne des chemins alloués.

spatiale des identifiants (cf. §2.4.1). Cependant, nous observons de petites différences entre les deux courbes résultant des comportements d'édition monotone.

Explication : Le comportement d'édition aléatoire conduit à un arbre exponentiel équilibré, d'où la croissance logarithmique des identifiants. Les autres cas d'édition sont très similaires car les sous-fonctions d'allocation ont la même probabilité d'apparaître. Les différences s'expliquent par les différents choix lors des exécutions. La progression sous-linéaire s'explique grâce à l'augmentation d'arité de l'arbre lorsqu'il gagne en profondeur, i.e., chaque nœud de l'arbre peut posséder jusqu'à deux fois plus de fils que son parent. Lorsque la sous-fonction d'allocation employée ne s'accorde pas avec le comportement d'édition, la profondeur de l'arbre est incrémentée rapidement. Lorsque finalement la bonne sous-fonction d'allocation est trouvée, celle-ci couvre les dépenses qu'il a fallu déployer pour la trouver, i.e., la perte de niveaux dans l'arbre.

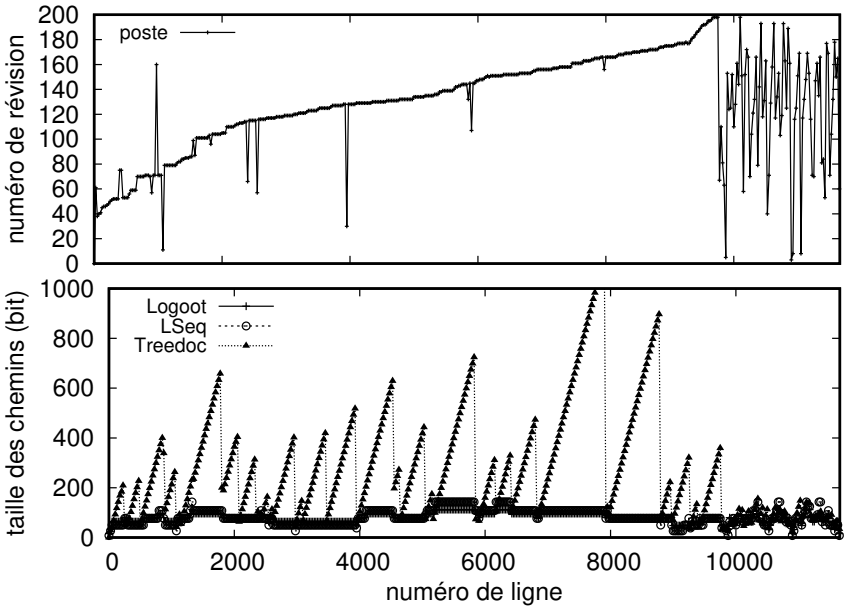
2.5.5 Complexité spatiale

2.5.5.a Traces réelles

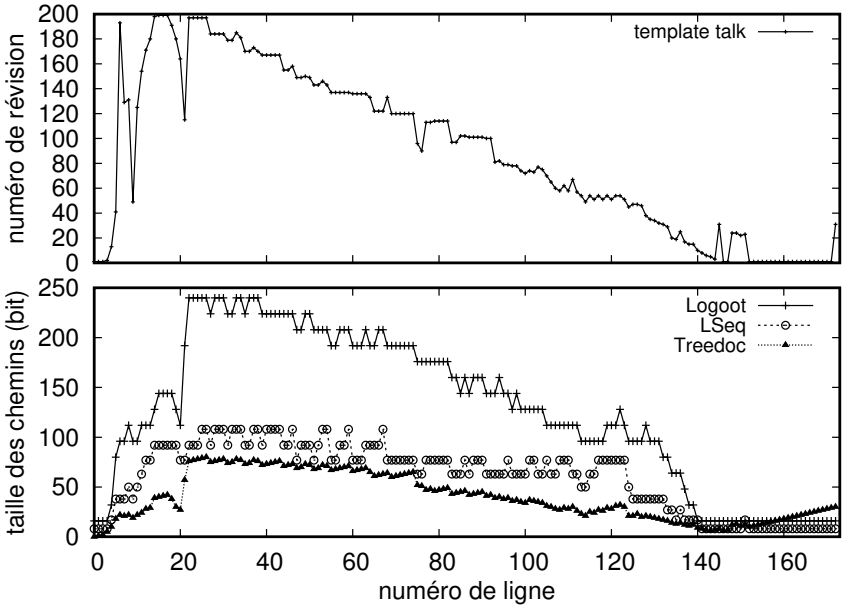
L'encyclopédie Wikipédia [101] répertorie des millions d'articles écrits collaborativement. Un utilisateur peut lire un article et, s'il le souhaite, en modifier le contenu. Lorsque ses modifications sont achevées, il les soumet à Wikipédia. Deux issues possibles : (i) La contribution est acceptée et sera visible de tous ou (ii) la contribution est rejetée car un autre utilisateur a effectué une modification en concurrence et l'a soumise en premier. Il faut alors réviser la version rejetée afin de l'adapter à la version la plus à jour avant de la soumettre à nouveau, si nécessaire. Wikipédia garde l'historique des modifications apportées à tous les articles depuis leur création. Nous sommes alors à même de rejouer les éditions – nommées révisions – dans l'ordre où elles ont été effectuées. Toutefois, toute trace de concurrence est effacée par le protocole d'édition même. D'autre part, la granularité est fixée à la ligne et non au caractère. En cela, les simulations sur corpus Wikipédia diffèrent légèrement de la réalité.

Objectif : Montrer que ni Logoot ni Treedoc ne parviennent à fournir des identifiants dont la taille soit satisfaisante quel que soit le document créé grâce à des traces réelles. En revanche, LSeq y parvient.

Description : Logoot est configuré avec une arité maximale de 2^{16}. LSeq est configuré avec arité maximale de départ de 2^8. Treedoc a une arité de 2^1 et est configuré pour utiliser sa méthode originelle peu adaptée à l'édition monotone – son autre stratégie étant équivalente à la stratégie de Logoot. Les documents considérés sont des articles extraits de Wikipédia et rejoués sur 200 révisions. L'un des articles atteint jusqu'à 10k lignes princi-



(a) Document Wikipédia de très grande taille principalement édité de gauche à droite.



(b) Document Wikipédia de petite taille principalement édité de droite à gauche.

FIGURE 2.18 – Taille du chemin alloué pour chaque ligne du document. L'axe des abscisses montre le numéro de la ligne concernée. L'axe des ordonnées de la partie haute de la figure montre l'âge de la ligne. L'axe des ordonnées de la partie basse de la figure montre la taille binaire du chemin alloué.

palement ajoutées de gauche à droite[2]. L'autre article atteint seulement 200 lignes mais est principalement édité de droite à gauche[3].

Résultat : Les figures 2.18a et 2.18b montrent la taille de l'identifiant associé à chaque ligne. Nous observons que Treedoc possède des chemins qui augmentent très vite quel que soit le type d'édition. Lorsque le nombre d'insertions successives est très grand (cf. figure 2.18a) les chemins atteignent des tailles très élevées avec une claire croissance linéaire. Dans ce cas, Logoot se comporte mieux. Bien que LSeq démarre avec une base plus faible il présente des résultats équivalents à Logoot. En revanche, dans le cadre de l'édition de droite à gauche (cf. figure 2.18b), Logoot alloue des chemins dont la taille augmente extrêmement rapidement. Treedoc se comporte de manière similaire bien que les valeurs mesurées soient plus faibles. Les identifiants alloués par LSeq se stabilisent.

Explication : Dans les deux types d'édition, Treedoc et Logoot allouent des identifiants dont la complexité est linéaire. Ainsi, plus les insertions se succèdent, plus l'arbre est déséquilibré, plus la taille du chemin augmente. Cependant, comme Treedoc alloue des chemins où chaque pas est binaire ($\mathcal{P} \in \mathbb{N}_{<2}.\mathbb{N}_{<2}\ldots\mathbb{N}_{<2}$), les chemins restent plus petits que ceux de Logoot dans le cadre du document édité de droite à gauche. D'un autre côté, Logoot a conçu sa stratégie pour l'édition de gauche à droite. Dès lors, si le comportement suit cette hypothèse, les identifiants grossissent par palier – linéairement certes, mais lentement. LSeq ne favorise aucun comportement d'édition et adapte la taille de ses identifiants à la taille du document. Ainsi, les deux types de document sont gérés : beaucoup d'insertions effectuées de gauche à droite ; peu d'insertions effectuées de droite à gauche.

2.5.5.b Documents synthétiques

Objectif : Montrer la progression de la taille des identifiants alloués par les fonctions d'allocation Logoot, LSeq et Treedoc sur des documents synthétiques.

Description : Deux types de document synthétique sont créés. Tout d'abord, un premier document est créé grâce à des insertions à des positions aléatoires dans la séquence. Le second document est créé grâce à des insertions successives en fin de document. La fonction d'allocation des chemins de Logoot est spécifiquement conçue pour gérer ce dernier type d'édition. Logoot est configuré avec une arité maximale de 2^{16}, LSeq est configuré avec une arité maximale de départ de 2^8, Treedoc a une arité maximale de 2^1. La taille moyenne des chemins composant les identifiants est mesurée à chaque nouvelle insertion. Le document atteint 500k caractères.

[2] https://fr.wikipedia.org/wiki/Liste_des_bureaux_de_poste_français_classés_par_oblitération_Petits_Chiffres

[3] https://en.wikipedia.org/wiki/Template_talk:Did_you_know

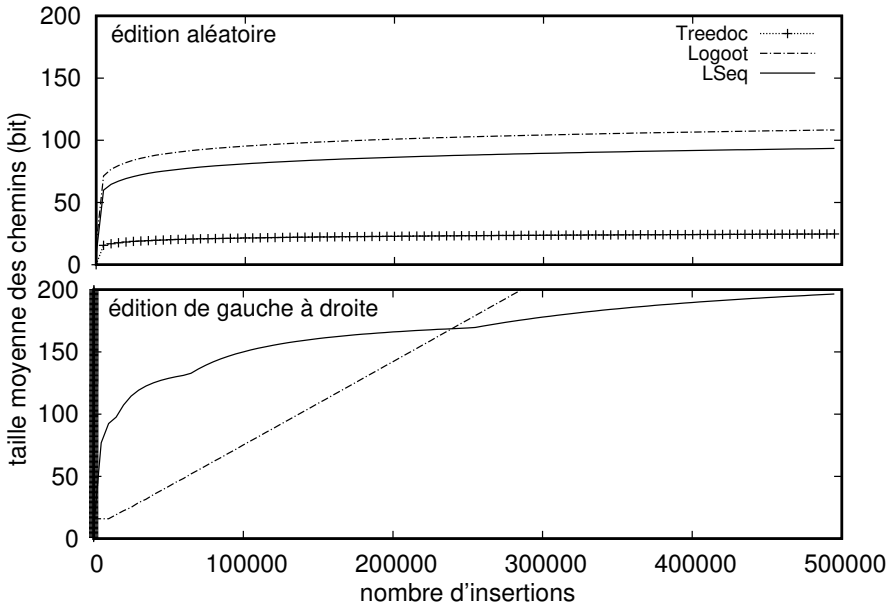


FIGURE 2.19 – Taille moyenne des chemins alloués en fonction du nombre d'insertions successives dans la séquence. L'axe des abscisses montre le nombre d'insertions dans la séquence. L'axe des ordonnées montre la taille moyenne de la représentation binaire des chemins alloués.

Résultat : La figure 2.19 montre les résultats de cette simulation. La partie du haut montre les résultats du document édité à des positions aléatoires. La partie du bas montre les résultats du document édité à la fin. Nous observons que l'édition à des positions aléatoires entraîne une génération de chemins dont la taille est logarithmique, que ce soit pour Logoot, LSeq ou Treedoc. Dans ce cas, Treedoc est meilleur que Logoot et LSeq. Nous observons aussi que l'édition de gauche à droite entraine une croissance extrêmement rapide pour les identifiants alloués par Treedoc (la courbe est superposée à l'axe des ordonnées). De son côté, Logoot alloue des chemins dont la taille augmente moins rapidement. Malgré cela, la croissance reste linéaire par rapport au nombre d'insertions dans la séquence. À terme, l'exécution d'un protocole de relocalisation des chemins devient nécessaire pour retrouver de bonnes performances. En revanche, LSeq alloue des identifiants dont la croissance est sous-linéaire par rapport au nombre d'insertions. À terme, l'allocation de LSeq finit par surpasser celle de l'état de l'art.

Explication : Les fonctions d'allocation Logoot, LSeq et Treedoc utilisent une structure d'arbre pour allouer le chemin associé à chaque caractère. Dans le cadre de l'édition aléatoire, l'arbre reste équilibré au cours des insertions. Les branches les plus basses se remplissent au maximum de leur capacité. Par conséquent, les identifiants alloués par ces trois approches croissent logarithmiquement par rapport au nombre d'insertions. L'arité de l'arbre de Treedoc étant inférieure à l'arité de l'arbre de Logoot et LSeq, Treedoc propose de meilleurs identifiants. Dans le cadre de l'édition de gauche à droite, chaque nouvelle insertion dans Treedoc ajoute un bit au chemin généré. La croissance est linéaire par rapport au nombre d'insertions dans la séquence. La fonction d'allocation de Logoot est conçue pour gérer ce type d'édition. Chaque niveau de l'arbre possède une branche remplie d'éléments. Toutefois, le nombre d'éléments accueillis par chaque branche reste constant, d'où la croissance linéaire de la taille des chemins générés. Avec LSeq, l'arbre est exponentiel : chaque nœud de l'arbre possède une arité maximale deux fois supérieure à son parent. Ainsi, plus elles sont profondes, plus les branches peuvent accueillir d'éléments. Par conséquent, la vitesse d'augmentation de profondeur de l'arbre exponentiel diminue lorsque le nombre d'insertions augmente.

2.5.6 Complexité temporelle

Objectif : Confirmer l'analyse en complexité temporelle de LSeq. Des performances passant à l'échelle sont attendues.

Description : Cette expérience implique un utilisateur artificiel unique effectuant des opérations sur sa réplique du document. Le banc d'essai s'est déroulé sur un *MacBook Pro* comportant un processeur *2.5 GHz Intel Core i5*, la version 4.1.1 de *Node.js*, sur une architecture Darwin 64-bit. Pour chaque opération, un document de $I - 1$ caractères est créé artificiellement. Les mesures s'effectuent sur la $I^{\text{ème}}$ opération. Les opérations prises en

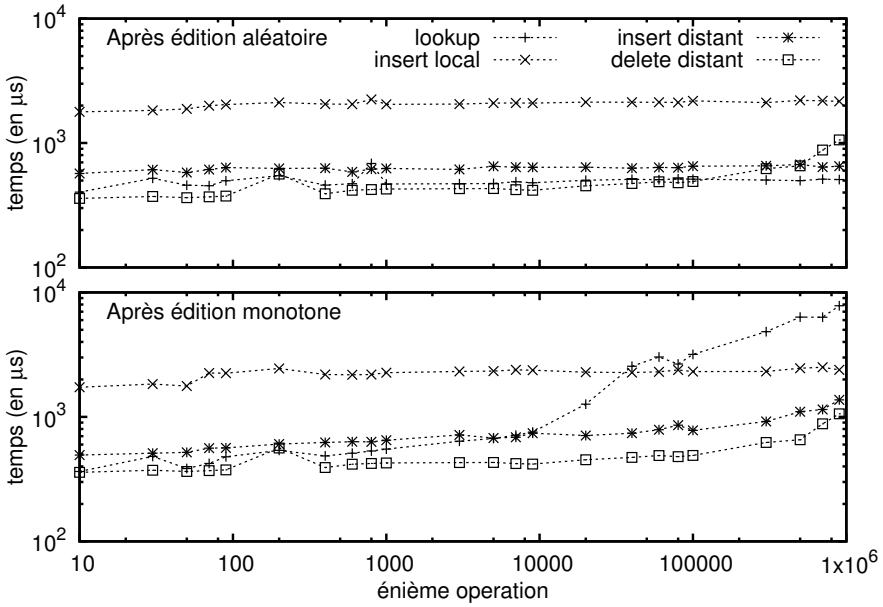


FIGURE 2.20 – Performances de la énième opération effectuée sur LSEQ. L'axe des abscisses montre le numéro de la énième opération. L'axe des ordonnées montre le temps d'exécution de l'opération en microsecondes. La figure du haut considère une structure remplie par des insertions aléatoires. La figure du bas considère une structure remplie par insertions monotones.

compte sont la fonction LOOKUP, la partie locale de l'insertion, la partie distante de l'insertion, et la partie distante de la suppression – la partie locale de cette dernière ne comprend que la propagation de l'identifiant ciblé aux autres membres. Les mesures sont effectuées à plusieurs reprises sur deux types de document. Tout d'abord, un document généré par un comportement d'édition aléatoire, i.e., l'arbre LSEQ est équilibré. Ensuite, un document généré par un comportement d'édition monotone, i.e., une seule branche de l'arbre LSEQ est remplie. Le langage JavaScript opère de nombreuses optimisations à la volée. Afin de montrer la contribution réelle de chacune des opérations, ces optimisations ont été limitées au maximum. Les performances réelles des opérations sont donc nettement meilleures que celles présentes dans cette expérimentation.

Résultat : La figure 2.20 montre les résultats de cette expérimentation. Nous observons que les valeurs mesurées, dans le cadre d'un arbre remplies par des éditions aléatoires, ne croissent pratiquement pas et ce quelle que soit l'opération. D'un autre côté, lorsque l'insertion locale après édition monotone reste stable, nous observons une croissance linéaire du temps d'exécution de la fonction LOOKUP et une plus faible évolution pour les parties distantes des opérations d'insertion et de suppression.

Explication : Après un comportement d'édition aléatoire, l'arbre de LSEQ est équilibré. Par conséquent l'influence d'une opération se limite à un petit sous ensemble d'éléments composant le document. Par exemple, la fonction LOOKUP n'a pas besoin d'explorer chaque élément de l'arbre. Elle rejette rapidement de nombreuses branches sans importance à chaque niveau de l'arbre car l'indice recherché ne tombe pas dans leur intervalle. Toutefois, cette remarque n'est pas vraie en ce qui concerne un arbre rempli par un comportement d'édition monotone. En effet, dans ce cas, la plupart des éléments sont localisés dans l'une – et la plus profonde – des branches de l'arbre. Ainsi, la fonction LOOKUP va probablement explorer tous les niveaux et inspecter chaque élément pour en compter le nombre de fils afin d'actualiser son indice de progression courant. Les mesures concernant les opérations distantes suivent le même raisonnement. Elles sont plus efficaces car l'exploration des niveaux utilise la recherche dichotomique.

2.5.7 Concurrence

Objectif : Montrer que plus la concurrence est élevée, plus les identifiants alloués sont petits.

Description : Deux utilisateurs artificiels dont le comportement est monotone écrivent de gauche à droite un même document. Grâce à netem [33], une latence est ajoutée sur le transport des messages créant des opérations concurrentes. Les documents atteignent 10k caractères. Les mesures sur la taille moyenne des messages envoyés sont effectuées tout au long de l'expérimentation. La fonction d'allocation employée est celle de LSEQ : un arbre exponentiel avec deux sous-fonctions d'allocation.

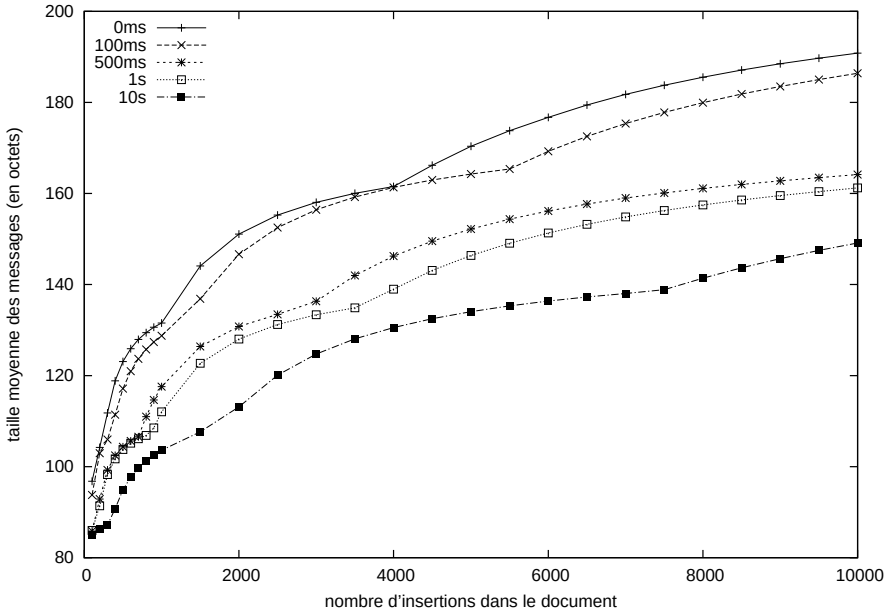


FIGURE 2.21 – Effets de la concurrence sur la taille des identifiants. L'axe des abscisses montre le nombre d'insertions effectuées dans le document. L'axe des ordonnées montre la taille des messages envoyés (en octets).

Résultat : La figure 2.21 montre les résultats de cette expérimentation. Comme attendu, plus la latence est élevée, plus la concurrence est élevée, plus les messages sont de petite taille. Les expérimentations sans concurrence présentent donc la borne supérieure des fonctions d'allocation. La figure 2.21 nous montre aussi, avec une plus fine granularité, la croissance polylogarithmique de la taille des messages qui confirme une seconde fois l'analyse en complexité spatiale des identifiants.

Explication : Les insertions effectuées par les participants ciblent une même position dans le document. Toutefois, ils ne prennent connaissance de cela que quelque temps plus tard, lorsque le message correspondant leur est parvenu. Ainsi, les opérations sont réellement concurrentes. Les insertions étant effectuées en concurrence à la même position dans la séquence, les opérations partagent le même espace d'allocation. Les identifiants ont une chance d'obtenir le même chemin dans l'arbre. En moyenne, ils sont plus rapprochés les uns des autres. Il est important de noter que les documents résultant de ces expérimentations sont différents. En d'autres termes, la série de caractères composant le document est différente en fonction de la latence.

2.6 Conclusion

Dans ce chapitre, nous avons présenté LSeq, une fonction d'allocation d'identifiants pour les structures de données répliquées sans conflits conçues pour les séquences et n'utilisant pas de pierres tombales.

L'utilisation d'un arbre exponentiel en tant que structure permet d'améliorer l'allocation sur des comportements d'édition au prix d'un pire cas plus onéreux. Ce pire cas est rendu difficile à atteindre grâce à l'utilisation de deux sous-fonctions d'allocation conçues pour les comportements d'édition monotone. Si malgré tout un utilisateur malintentionné tente d'obtenir de tels identifiants, la différence entre les identifiants attendus et les identifiants incriminés est telle qu'il est facile de confondre le coupable et donc de prendre des mesures à son encontre. La complexité des identifiants LSeq est bornée par un polylogarithme par rapport au nombre d'insertions effectuées dans la séquence. LSeq permet donc de construire un éditeur collaboratif pouvant gérer des documents de grande taille rédigés par de nombreux participants.

Tout comme les approches basées sur le maintien de répliques distantes, LSeq a besoin d'un protocole de dissémination fiable de messages. En effet, les identifiants générés et supprimés par LSeq doivent parvenir à toutes les répliques pour que celles-ci convergent vers un état équivalent. Le chapitre suivant présente un protocole construisant un réseau superposé : chaque serveur détenant une réplique connaît un ensemble de serveurs détenant également une réplique du même document et peut communiquer avec ceux-ci. La spécificité du protocole présenté est qu'il maintient ces ensembles de telle sorte que le cardinal de chaque ensemble croît et décroît logarithmiquement par rapport à la taille du réseau et

ce sans connaissance globale. Grâce à ce protocole, un éditeur collaboratif est en mesure de disséminer efficacement les changements effectués dans le document à tous les autres éditeurs collaboratifs impliqués dans la rédaction (cf. §4).

3

Un protocole d'échantillonnage aléatoire adaptatif

Sommaire

Les éditeurs collaboratifs répartis [26] suivent le schéma de réplication optimiste [23, 52, 55, 88]. Pour que toutes les répliques d'un document convergent vers un état équivalent, toute opération générée sur une réplique doit parvenir à toutes les autres répliques afin d'être intégrée. Un mécanisme de diffusion fiable, décentralisé et compatible avec le Web est requis.

Grâce à WebRTC [45], il est dorénavant possible d'établir des connexions de navigateur à navigateur et par conséquent d'implémenter un tel mécanisme de diffusion. Les mécanismes de diffusion se basent sur des protocoles d'échantillonnage de pairs. Toutefois, déployer ces protocoles avec WebRTC n'est pas sans poser de problèmes :

(i) WebRTC ne gère ni les adresses ni les routes ce qui rend l'établissement de connexion plus coûteux que sur un réseau IP et sujet aux défaillances.

(ii) Les navigateurs Web fonctionnent sur des outils informatiques aux capacités hétérogènes tels que les ordinateurs de bureau, les téléphones portables ou les tablettes tactiles. Le protocole doit donc réduire sa consommation en ressources au maximum.

(iii) Avec WebRTC, un simple lien HTTP permet de partager l'accès à une session d'édition en temps réel. Cette simplicité d'accès expose l'éditeur aux explosions soudaines de popularité caractéristiques du Web. Par exemple, un lien « tweeté » par une célébrité sur un événement particulier peut attirer des milliers de participants en très peu de temps. Cette soudaine popularité retombe lorsque l'événement s'achève. Le mécanisme de diffusion et le protocole duquel il dépend doivent s'adapter à ces montées fulgurantes en nombre de participants et maintenir leur qualité de service avant de revenir à leur configuration initiale lorsque le pic de popularité retombe.

Malheureusement, les protocoles d'échantillonnage actuels ne parviennent pas à répondre à l'ensemble de ces problématiques. D'un côté, Scamp [35, 36] propose une forme d'adaptation en fournissant une vue dont la taille est $\ln(N) + k$, N étant le nombre de membres dans le réseau et k une constante positive. Cependant, sa procédure d'établissement de connexion impliquant des propagations aléatoires systématiques n'est pas adapté au contexte WebRTC. D'une autre côté, les protocoles plus utilisés tels que Cyclon [99] fournissent une vue dont la taille est constante et paramétrée lors du déploiement. De ce fait, le développeur est obligé de surdimensionner la taille des vues afin de gérer les possibles pics de popularités.

Ce chapitre présente Spray [77], un protocole dont les pairs voient leur vue partielle s'ajuster à la taille du réseau, sans l'usage d'aucune connaissance globale. Le cycle de vie d'un pair, divisé entre son entrée, sa vie et son départ, est conçu pour conserver un nombre d'arcs dans le système en rapport avec la taille de ce dernier. Les protocoles construits au dessus de Spray peuvent bénéficier de cet ajustement automatique. Ainsi, le trafic généré par un protocole de dissémination de messages peut évoluer selon les dimensions du réseau.

Ce chapitre commence par présenter l'état de l'art des protocoles d'échantillonnage de pairs en les divisant en deux catégories : ceux dont les vues partielles sont fixes, et ceux dont les vues partielles s'adaptent au réseau. La section 3.2 définit le problème scientifique. La section 3.3 présente Spray, un protocole appartenant à la seconde catégorie. La section 3.4 présente et compare les propriétés des protocoles d'échantillonnage. La section 3.5 présente l'effet de Spray sur un protocole de dissémination épidémique de messages. La section 3.6 conclut le chapitre.

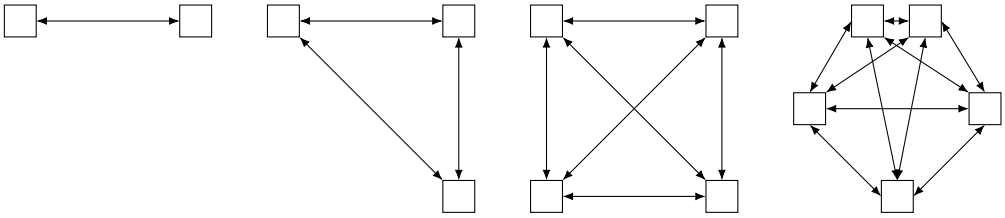


FIGURE 3.1 – Graphes complets.

3.1 État de l'art

Un réseau superposé (*overlay network*) permet à un ensemble de machines de communiquer entre elles au dessus d'un autre réseau tels que, par exemple, l'internet ou un autre réseau superposé. Toutes sortes de réseaux superposés existent. Si chaque machine possède un canal de communication avec chacune des autres machines, alors la topologie correspond à celle d'un graphe complet (cf. figure 3.1). La progression quadratique du nombre d'arcs en fonction du nombre de nœuds empêche un tel système d'atteindre de grandes envergures. Afin de résoudre ce problème, les machines ne possèdent qu'une vue partielle du réseau. Cette vue partielle constitue le voisinage direct d'un nœud. Le défi consiste alors à peupler ces vues avec des liens logiques selon les critères souhaités.

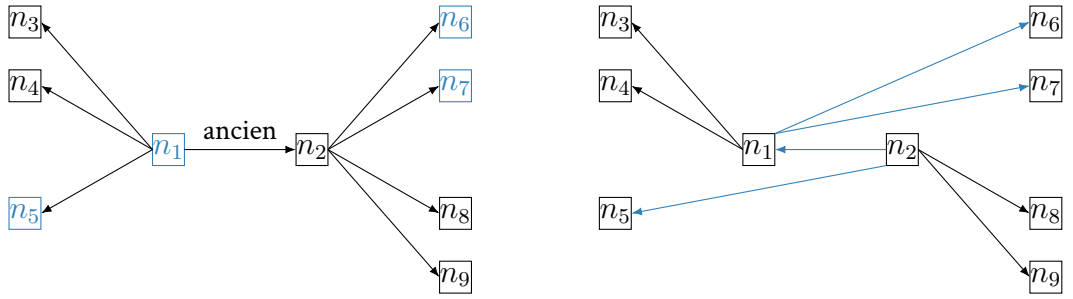
Dans ce manuscrit, nous nous intéressons aux protocoles d'échantillonnage aléatoire de pairs [50] dont l'objectif est de fournir à chaque machine une vue partielle peuplée d'adresses logiques réparties selon une loi uniforme. La topologie en résultant est proche de celle des graphes aléatoires [28]. Ces réseaux possèdent d'intéressantes propriétés tels que la tolérance aux pannes ou un faible diamètre. La première garantit que lorsqu'un nœud disparaît soudainement le graphe reste connecté avec une forte probabilité, la seconde garantit que les messages se propagent rapidement aux membres du réseau.

Deux familles d'approches existent quant à la taille des vues partielles : (i) la famille d'approches dont la taille est fixée à la configuration (cf. §3.1.1) ; (ii) la famille d'approches dont les nœuds, lorsqu'ils rejoignent le réseau, contribuent à hauteur du logarithme de la taille du réseau (cf. §3.1.2).

3.1.1 Taille fixe

Les protocoles d'échantillonnage aléatoire de pairs peuplant des vues dont la taille est constante possèdent un socle commun à savoir l'échange périodique de voisinages.

CYCLON [99]. CYCLON est un protocole où chaque nœud possède une vue partielle dont chaque arc est associé à un âge. Régulièrement, les nœuds initient un mélange (*shuffling*)



(a) L'initiateur du mélange choisit d'envoyer n_5 et lui-même à son plus vieux voisin n_2. En retour, il lui propose n_6 et n_7.

(b) Les arcs sont échangés de part et d'autre.

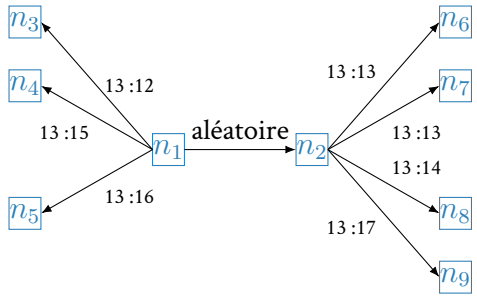
FIGURE 3.2 – Exemple de mélange par CYCLON. Pour améliorer la lisibilité, seuls les vues partielles du n_1 et n_2 sont explicitées.

avec leur voisin le plus âgé. Si la communication est possible, le mélange est amorcé et l'âge de tous les voisins est incrémenté, sinon l'adresse est supprimée. Cela permet de supprimer les nœuds considérés comme partis.

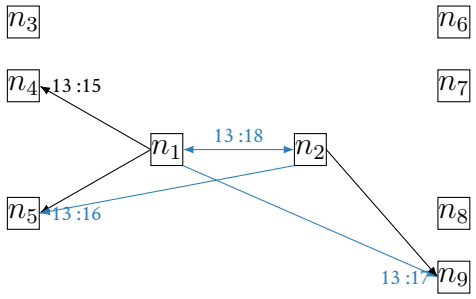
Lors de ce mélange, le nœud envoie un nombre prédéfini de ses arcs au nœud choisi, en prenant soin de remplacer l'arc vers son homologue par un arc vers lui-même. Les adresses sont choisies aléatoirement. À la réception, son vis-à-vis choisit un nombre d'arcs équivalent et les lui envoie. Les deux nœuds intègrent alors l'ensemble qu'ils ont reçu. Lorsque l'ensemble obtenu est trop grand pour la taille de vue partielle prédéfinie, des arcs sont rejetés en conservant de préférence les nouveaux arcs reçus et en supprimant les doublons.

La figure 3.2 décrit un exemple de mélange initié par le nœud n_1. Dans cet exemple, les vues partielles sont configurées pour accueillir 4 arcs et en échanger 2 pendant les mélanges. La figure 3.2a montre que n_1 choisit son plus vieux voisin afin d'initier l'échange, à savoir n_2. Il incorpore dans l'échange sa propre identité ainsi que celle d'un arc choisi aléatoirement. Le nœud n_2 reçoit la demande de mélange et choisit 2 nœuds aléatoirement, ici n_6 et n_7 qu'il envoie à son tour. La figure 3.2b montre que les nœuds participant au mélange ont supprimé les arcs qu'ils ont envoyé et ajouté ceux nouvellement reçus. En particulier, l'inversion de l'arc ayant permis le mélange garantit que le graphe reste connexe.

Newscast [97]. Newscast est un protocole où les vues partielles associent à chaque arc une estampille (*timestamp*). Les nœuds de Newscast effectuent un mélange périodiquement. Pour cela, ils choisissent un de leurs voisins aléatoirement et envoient la totalité de leur vue partielle à laquelle est ajoutée leur propre identité associée à une estampille à jour. Lors de la réception, leur homologue en fait de même. Tout deux fusionnent leurs tables de voisinage de telle sorte que seules les entrées les plus récentes selon l'estampille sont conservées.



(a) L'initiateur du mélange choisit n_2 aléatoirement et lui envoie sa vue à laquelle il s'ajoute avec une estampille à jour. À la réception, n_2 en fait de même.



(b) Les arcs sont échangés de part et d'autre en conservant les estampilles les plus récentes.

FIGURE 3.3 – Exemple de mélange par Newscast. Ici, les estampilles sont simplement formatées $hh : mm$.

Un nœud qui quitte le réseau n'effectue plus ce mélange périodique et ainsi n'annonce plus sa présence au reste du réseau. Petit à petit, les vues partielles possédant un arc pointant vers ce nœud s'en débarrassent car l'arc est devenu obsolète en comparaison des arcs nouvellement reçus.

Les pertes de message dues au manque de fiabilité des communications mettent en danger la distribution des arcs. Par exemple, si un message sur deux est perdu lorsqu'un certain nœud tente de communiquer, il perd autant d'occasions de s'annoncer au réseau. Par conséquent, il encourt le risque d'être évincé de certaines vues partielles. Dans le pire cas, il disparaît complètement. Newscast propose de compenser cela. Ainsi, si un nœud perd un message sur deux, il émet deux fois plus. La distribution des arcs s'en trouve ajustée.

La figure 3.3 montre un exemple de mélange avec le protocole Newscast. La taille des vues partielles est fixée à 4 entrées. Le nœud n_1 initie le mélange avec un voisin choisi aléatoirement : n_2. Il envoie l'intégralité de sa vue à laquelle il s'ajoute avec l'estampille actuelle $13 : 18$. Le nœud n_2 le reçoit et en fait de même. Les deux nœuds n_1 et n_2 conservent les quatre arcs les plus récents qu'ils possèdent, à savoir les arcs vers leur homologue, n_4, n_5 et n_9. Dans la figure 3.3b, nous remarquons que le réseau semble collapser. En particulier, certains nœuds ne sont plus référencés ni par n_1, ni par n_2. Toutefois, il ne s'agit là que de la représentation locale au mélange. Globalement, la topologie résultante reste proche de celle des graphes aléatoires.

Lpbcast [29]. Lpbcast est le diminutif de *lightweight probabilistic broadcast*. C'est un protocole dont les vues partielles ne comprennent que des arcs sans métadonnées additionnelles. Les vues sont rafraîchies lorsqu'un nœud diffuse un message au réseau. Chaque message comporte un ensemble borné d'identités de nœuds ayant quitté le réseau et un ensemble

borné d'identités de nœuds échantillonnés au cours du cheminement du message. Si aucun message n'a été reçu ni envoyé dans un certain intervalle de temps, une diffusion est automatiquement générée afin d'activer le mécanisme de mélange.

Lors de la réception de tels messages, un nœud peuple sa propre vue partielle des nouveaux éléments présents dans le message avant de la réajuster à la taille maximale autorisée en supprimant aléatoirement des arcs.

HyParView [59]. HyParView est un protocole possédant deux vues partielles. La première est active et est utilisée lors de l'émission de messages au réseau. La seconde est passive et sert à remplacer les arcs obsolètes de la vue active, e.g., lors d'une défaillance ou d'un départ volontaire. La vue active est considérablement plus petite que la vue passive et les arcs compris dans cette première sont vérifiés à chaque message émis.

Lors d'un mélange, les arcs contenus dans les deux vues sont sujets à échange dans le but de supprimer les nœuds défaillants de toutes les vues passives. Similairement à CYCLON, les arcs privilégiés sont ceux nouvellement reçus.

Les approches à taille fixe constituent le cœur de nombreux protocoles décentralisés [22, 30, 49, 54]. Malheureusement, leur manque d'adaptivité n'est pas compatible avec notre contexte. La section suivante introduit une autre famille de protocole d'échantillonnage aléatoire de pairs fournissant des vues partielles s'ajustant automatiquement à la taille du réseau.

3.1.2 Taille variable

La famille de protocoles d'échantillonnage aléatoire de pairs peuplant des vues dont la taille n'est pas définie au préalable a pour seul représentant :

SCAMP [35, 36]. SCAMP est l'acronyme de *SCAlable Membership Protocol*. Il s'agit d'un protocole d'échantillonnage associant une réaction à certains événements. En particulier, l'entrée dans le réseau doit permettre d'augmenter la taille moyenne des vues partielles. Cette augmentation est logarithmique. Le départ d'un nœud doit entraîner une équivalente diminution.

Les pairs utilisant SCAMP maintiennent deux vues partielles correspondants aux arcs entrant et aux arcs sortant.

Lorsque un nœud rejoint le réseau, il contacte un membre qu'il ajoute directement à sa vue partielle. Ce contact annonce ensuite le nouveau venu au réseau. Pour ce faire, il crée autant de messages d'annonce qu'il possède de voisins et les dissémine. Chaque nœud recevant ce type de message est libre d'accepter l'annonce en ajoutant alors l'identité du nouveau venu à sa vue partielle, ou la refuser, auquel cas il réexpédie le message à l'un de ces voisins. Afin de répartir équitablement les annonces, chaque nœud n_i a une probabilité inverse-

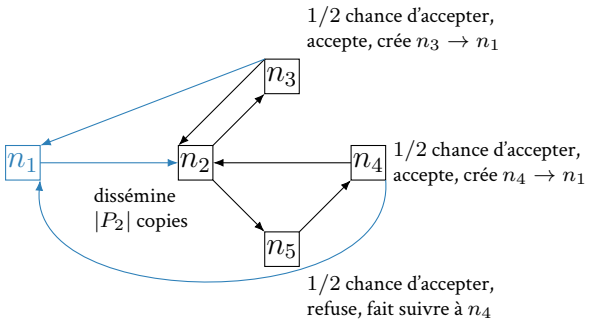


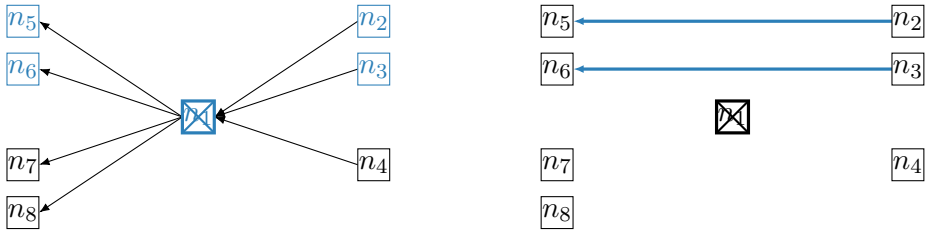
FIGURE 3.4 – Exemple de nœud rejoignant le réseau dans SCAMP.

ment proportionnelle à la taille de sa vue partielle d'accepter l'annonce : $1/(|P_i|+1)$. Cette répartition des arcs seuls permet de créer une topologie proche de celle des graphes aléatoires. Cependant, la vue partielle du nouveau nœud est faiblement peuplée au contraire des nœuds plus vieux. Cela met en danger la robustesse du réseau face aux défaillances.

La figure 3.4 montre l'entrée d'un nœud n_1 dans un petit réseau SCAMP composé des nœuds n_2, n_3, n_4 et n_5. Dans cet exemple, le nœud n_1 contacte n_2 et l'ajoute directement à sa vue partielle. n_2, possédant 2 voisins, copie l'identité de n_1 2 fois et les dissémine à n_3 et n_5. Lorsque n_3 reçoit le message, il possède une chance sur deux d'accepter l'annonce, ce qu'il fait : un arc est ajouté entre n_3 et n_1. Le nœud n_5 procède de la même façon. Toutefois, il n'accepte pas l'annonce et retransmet le message à son voisin n_4. Ce dernier l'accepte et l'arc allant de n_4 à n_1 est créé. Au total, 3 arcs ont été créés à l'arrivée du nœud n_1.

Lorsqu'un nœud souhaite quitter le réseau, (i) il œuvre à préserver la connectivité de ce dernier et (ii) il emporte avec lui une quantité d'arcs correspondant à l'entrée du dernier nœud dans le réseau. Pour ce faire, le nœud va agir comme pont entre les nœuds de sa vue partielle entrante et ceux de sa vue partielle sortante. Ainsi, il permettra de ressouder la maille du réseau qu'il aurait défait en partant. Ce mécanisme est généralement applicable à tous les protocoles d'échantillonnage mais il requiert de maintenir deux vues partielles – entrante et sortante – et des canaux de communication bidirectionnels. Il requiert aussi un travail supplémentaire de la part du nœud sortant impossible à garantir lors de défaillances : le nœud devra servir d'intermédiaire à la connexion de tous les voisins de sa vue partielle entrante (sauf un) vers autant de voisins appartenant à sa vue partielle sortante. Ainsi, le nombre d'arcs supprimés correspond en moyenne à celui injecté à l'entrée d'un nœud dans le réseau. Ce mécanisme améliore considérablement le maintien de la connexité du réseau mais ne représente pas une solution fiable pour autant. En effet, certains nœuds peuvent se trouver sans voisins dans l'une de leurs vues partielles.

La figure 3.5 présente le mécanisme activé lors du départ du nœud n_1 utilisant SCAMP. n_1 fournit à chaque nœud de sa vue partielle entrante l'identité d'un nœud présent dans sa vue partielle sortante. Ainsi, le nœud n_2 ajoute n_5 à sa vue partielle sortante, n_3 ajoute n_6 à sa



(a) Le nœud n_1 souhaite quitter le réseau. Il en informe ses vues partielles afin d'effectuer un pont entre eux.

(b) Le nœud n_1 notifie aux nœuds n_2 et n_3 qu'ils doivent ajouter n_5 et n_6 dans leur vue partielle sortante respective.

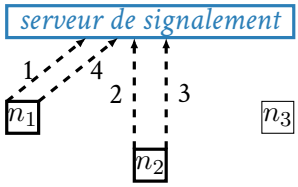
FIGURE 3.5 – Exemple de sortie de réseau dans SCAMP. Seules les vues partielles entrante et sortante de n_1 sont explicitées.

vue partielle sortante, n_4 n'ajoute rien. La figure 3.5b montre les arcs après le départ de n_1. Entre autres, nous observons que si n_4 ne possède pas d'autres voisins dans sa vue partielle sortante, ou si n_7 ou n_8 ne possèdent pas d'autres voisins dans leur vue partielle entrante, alors le réseau n'est plus connexe.

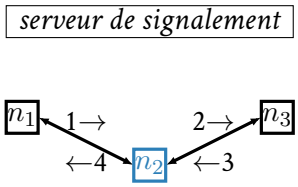
SCAMP possède la capacité d'ajuster les vues partielles de ses membres à la taille du réseau auquel ils appartiennent. Malheureusement, cette approche fait face au problème suivant :

Propagation des arcs. Les annonces sont systématiquement propagées à distance de plusieurs nœuds. Ce mécanisme participe à la construction rapide d'une topologie proche de celle des graphes aléatoires. Toutefois, entre l'origine et la destination de l'annonce, toute perte de message empêche l'établissement de la connexion. Cela s'avère particulièrement problématique lorsque l'établissement d'une connexion nécessite plusieurs allers-retours comme c'est le cas dans le contexte WebRTC [45] : cette technologie permet l'établissement de canaux de communication d'un navigateur Web à l'autre, et ce, même en présence de configurations réseaux complexes impliquant pare-feu, proxy, ou NAT (*Network Address Translation*). Toutefois, WebRTC ne gère ni l'adressage, ni le routage. Établir une connexion WebRTC requiert une négociation où le nœud d'origine et le nœud de réception s'envoient mutuellement des moyens d'accès distants dans l'ordre défini du plus aisé au plus ardu. Par exemple, les échanges vont d'abord concerner la boucle locale (*localhost*), puis le réseau local (e.g. $192.168.255.255$), puis l'internet etc. Aussitôt que la négociation s'achève avec succès, un canal de communication bidirectionnel est établi. Les nœuds peuvent alors communiquer entre eux.

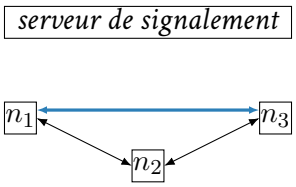
Pour établir une connexion, les navigateurs s'échangent des offres et acquittements via un médiateur commun (e.g. mails, services dédiés de signalement, connexions WebRTC connues, etc.). Dans la figure 3.6a, n_1 souhaite se connecter à n_2. Par conséquent, n_1 envoie ses offres au service de signalement connu. Le nœud n_2 récupère l'offre et envoie



(a) n_1 se connecte à n_2 via un médiateur. 1 : n_1 crée ses offres ; 2 : n_2 récupère ces offres ; 3 : n_2 crée ses offres en réponse ; 4 : n_1 : reçoit les offres et établit une connexion bidirectionnelle avec n_2. n_3 en fait de même avec n_2. La figure 3.6b décrit le réseau en résultant.



(b) n_1 se connecte à n_3 en utilisant n_2 comme médiateur. 1 : n_1 envoie ses offres à n_2 ; 2 : n_2 redirige les offres à n_3 ; 3 : n_3 envoie ses offres en réponse à n_2 ; 4 : n_2 redirige les offres vers n_1 qui se connecte à n_3.



(c) Le réseau superposé : Un réseau complètement connecté composé de 3 membres.

FIGURE 3.6 – Créer un réseau superposé au dessus de WebRTC.

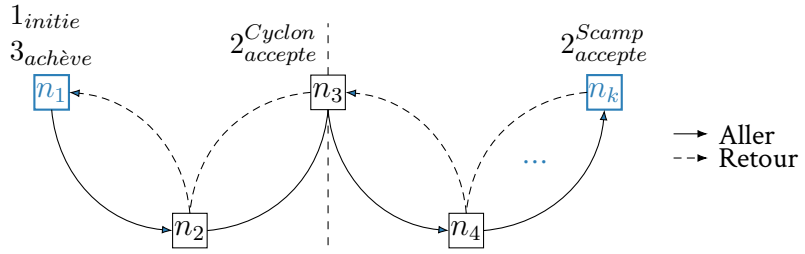


FIGURE 3.7 – Établissement d'une connexion impliquant des allers-retours avec WebRTC. (1) n_1 initie la création d'une connexion. Le message transite par n_2. (2) Lorsque CYCLON établit une connexion avec le voisin du voisin – ici n_3 – SCAMP cherche k voisins plus loin. (3) Un message est envoyé en sens inverse jusqu'à n_1 pour que la connexion soit finalement établie.

ses propres offres en réponse au service de signalement. Enfin, n_1 récupère les offres de n_2 et établit une connexion bidirectionnelle avec n_2. De manière identique, n_3 établit une connexion avec n_2. Désormais, le nœud n_1 est capable d'établir une connexion avec n_3 sans intermédiaire. Pour cela, il utilise n_2 comme médiateur. Toutefois, si le nœud n_2 tombe en panne durant cette procédure, la connexion ne pourra s'effectuer correctement, et ce, même si une route alternative existe (puisque WebRTC ne gère pas le routage).

La figure 3.7 montre la différence entre SCAMP et une approche établissant des connexions de voisin à voisin tel que CYCLON. Avec SCAMP, pendant l'établissement d'une connexion WebRTC, non seulement le message ne doit pas se perdre, mais les nœuds et arcs composant le chemin de l'annonce doivent rester disponibles pour préserver le pont entre les nœuds établissant une connexion. Soit P_f la probabilité qu'un élément (nœud ou arc) tombe en panne ou quitte le réseau pendant l'établissement d'une connexion. Soit P_E la probabilité que l'établissement d'une connexion soit achevé avec succès. Sans les allers-retours de WebRTC, la probabilité est :

$$P_E^{Scamp} = 1 - (1 - P_f)^{k+1} \tag{3.1}$$

Cela correspond à la probabilité que chaque élément sur le chemin de taille $k+1$ reste disponible et fonctionnel pendant leur tour de dissémination. Avec WebRTC, au minimum un aller-retour est nécessaire entre les nœuds se connectant. Aucun des éléments constituant le chemin de dissémination ne doit tomber en panne ou partir avant le retour du message. Nous obtenons :

$$\begin{aligned} P_{E,\,webrtc}^{Scamp} &= 1 - ((1 - P_f)^{2(k+1)}(1 - P_f)^{2k} \dots (1 - P_f)^2) \\ &= 1 - (1 - P_f)^{k^2+3k+2} \end{aligned} \tag{3.2}$$

En d'autres termes, le premier nœud et le premier arc doivent rester disponibles et fonctionnels $2k+2$ unités de temps. Le second nœud et le second arc doivent rester disponibles

et fonctionnels $2k$ unités de temps etc. Avec WebRTC, la probabilité d'échec sur l'établissement d'une connexion a augmenté d'une classe de complexité. Par conséquent, l'établissement d'une connexion suivant le protocole Scamp est particulièrement vulnérable aux défaillances. Le réseau perd rapidement sa connexité.

3.2 Définition du problème

À la lumière des différentes limitations des approches composant l'état de l'art, nous définissons le problème suivant :

Définition du problème 2. *Soit t une unité de temps arbitraire, soit $\mathcal{N}^t$ l'ensemble des membres non-byzantins du réseau à un instant t et soit P_i^t la vue partielle du nœud $n_i \in \mathcal{N}^t$. Un protocole d'échantillonnage aléatoire de pairs efficace doit assurer les propriétés suivantes :*

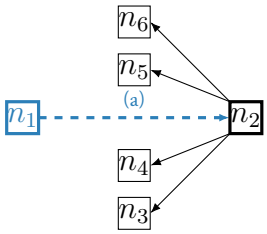
1. *Taille des vues partielles :* $\forall n_i \in \mathcal{N}^t, |P_i^t| \approx \mathcal{O}(\ln |\mathcal{N}^t|)$
2. *Établissement de connexion :* $\mathcal{O}(1)$

En d'autres termes, les vues partielles doivent s'adapter aux évolutions du réseau et rester équilibrées. Les approches à taille fixe échouent à fournir cette propriété. Le nombre de voisins parcourus doit être borné par une constante. Scamp échoue à fournir cette propriété puisque chaque connexion implique une dissémination aléatoire à une distance non bornée.

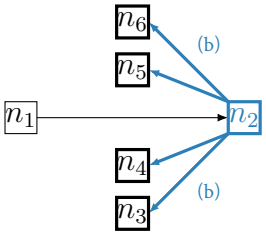
3.3 Spray : un protocole d'échantillonnage adaptatif

Spray est un protocole d'échantillonnage de pairs adaptatif inspiré à la fois de Scamp [36] et de Cyclon [99]. Spray comprend trois parties représentant le cycle de vie d'un nœud dans le réseau. Tout d'abord, la procédure pour rejoindre le réseau injecte un nombre logarithmique d'arcs par rapport à la taille du réseau. Ensuite, chaque nœud exécute une routine périodique dont le but est d'équilibrer les vues partielles en termes de taille et d'uniformité sur les nœuds référencés dans celles-ci. Le réseau converge rapidement vers une topologie possédant des propriétés similaires à celles des graphes aléatoires [28]. Enfin, lorsqu'un nœud tombe en panne ou quitte le réseau sans prévenir ses voisins, les propriétés du réseau restent stables.

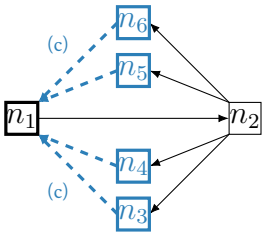
L'obtention de cette propriété d'adaptivité repose sur la conservation d'un nombre d'arcs approprié durant tout le cycle de vie du réseau superposé. En effet, contrairement à Cyclon, Spray est toujours à la limite du nombre optimal d'arcs. Comme Spray n'ajoute jamais d'arcs après la procédure d'entrée, toute suppression d'arc est définitive. De telles décisions ne sont donc pas prises à la légère. Ainsi, dans un premier temps, Spray ajoute des arcs lors de l'entrée d'un nœud dans le réseau. Dans un second temps, la procédure périodique de



(a) Le nœud n_1 contacte n_2 afin de rejoindre le réseau. n_1 ajoute n_2 à sa vue partielle.



(b) L'événement onSubs(n_1) est déclenché en n_2 qui dissémine l'annonce.



(c) L'événement onFwdSubs(n_1) est déclenché en n_{3-6}. Chacun de ces nœuds ajoute n_1 dans sa vue partielle.

FIGURE 3.8 – Exemple de procédure d'entrée dans le réseau de SPRAY.

mélange des arcs de SPRAY préserve tous les arcs du réseau. Dans un troisième temps, la procédure de sortie supprime précautionneusement quelques arcs. Dans l'idéal, il s'agit du nombre d'arcs ajoutés par le dernier nœud entré dans le réseau.

Parfois, conserver un nombre d'arcs global constant force les procédures de mélange et de départ à créer des doublons dans les vues partielles. Ainsi, une vue partielle peut contenir plusieurs fois le même voisin. Toutefois, ces doublons restent peu nombreux. De ce fait, ils n'ont pas d'impact notable sur la connexité du réseau.

Cette section se décompose en trois parties représentant le cycle de vie d'un nœud : la procédure d'entrée (cf. §3.3.1), les mélanges (cf. §3.3.2), la gestion des départs (cf. §3.3.3).

3.3.1 Rejoindre le réseau

La procédure d'entrée d'un nœud dans le réseau est la seule manière d'introduire de nouveaux arcs. Afin de répondre à la première partie de l'énoncé du problème 2, ce nombre d'arcs doit augmenter logarithmiquement par rapport à la taille du réseau. Tout comme dans SCAMP, nous supposons que chacun des nœuds respecte cette contrainte. Dès lors, ces derniers utilisent cette supposition afin de propager l'identité du nouvel arrivant. L'algorithme 3 décrit la façon dont le contact propage cette annonce à son voisinage où elle

est directement intégrée à leur vue partielle. En définitive, le nombre d'arcs dans le réseau augmente de $1 + \ln|\mathcal{N}|$ et ce, seulement en utilisant des interactions de voisin à voisin. Globalement, cela correspond à une augmentation de l'ordre de $|\mathcal{N}| \ln|\mathcal{N}|$ arcs, similaire au seuil de connexité des graphes aléatoires [28].

Algorithme 3 Procédure d'entrée de SPRAY.

```
1 : INITIALLY :
2 :     P ← ∅ ;                                  ▷ la vue partielle est un multiensemble
3 :     p ;                                      ▷ identité du nœud local
4 :
5 : EVENTS :
6 :     function onSubs(o)                       ▷ o : origine
7 :         if (|P| = 0) then
8 :             onFwdSub(o) ;                    ▷ cas particulier : réseau de 2 nœuds.
9 :         else
10 :            for each ⟨q, _⟩ ∈ P do sendTo(q, 'fwdSubs', o) ;
11 :        end if
12 :    end function

13 :    function onFwdSubs(o)                    ▷ o : origine
14 :        P ← P ⊎ {⟨o, 0⟩} ;
15 :    end function
```

L'algorithme 3 présente les instructions de SPRAY concernant l'entrée d'un nouveau nœud dans le réseau. La vue partielle est un multiensemble de couples $\langle n, age\rangle$ qui associe, à chaque voisin, son âge dans la vue partielle. Ce multiensemble permet de gérer les doublons. Ces doublons ont un impact négatif du fait de la redondance d'information. Toutefois, ils nous sont nécessaires afin de ne pas perdre d'arcs. L'âge joue le même rôle que dans CYCLON, c'est-à-dire qu'il accélère la suppression des nœuds avec lesquels il n'est plus possible de communiquer. L'événement onSubs est déclenché chaque fois qu'un nœud rejoint le réseau via ce contact. Ce dernier distribue l'identité du nouvel arrivant à son voisinage, plusieurs fois aux doublons, et indépendamment de l'âge. L'événement onFwdSubs se déclenche lorsqu'un nœud reçoit l'annonce d'un nouveau membre propagée par le contact. Le nœud ajoute alors l'identité reçue à sa vue partielle avec un âge initialisé à 0.

La figure 3.8 décrit un scénario où le pair n_1 contacte le nœud n_2 afin de rejoindre le réseau composé de $\{n_2, n_3, n_4, n_5, n_6\}$. Pour simplifier, la figure ne montre que les arcs nouvellement introduits ainsi que le voisinage respectif de n_1 et n_2. Le nœud n_1 ajoute directement n_2 dans sa vue partielle. Ce dernier redirige l'identité de n_1 à chacun de ses voisins. Chacun de ces voisins ajoute alors n_1 à leur vue partielle. Au total, SPRAY établit 5 connexions. Le réseau en résultant est connexe.

Malheureusement, les vues partielles des derniers arrivants sont clairement déséqui-

librées comparées à celles des membres plus anciens. De ce fait, ils violent la première condition de l'énoncé du problème 2. La section suivante décrit la procédure périodique de mélange dont l'objectif consiste à équilibrer les vues partielles.

3.3.2 Mélanger son voisinage

À la différence de CYCLON, SPRAY mélange des vues partielles dont les tailles peuvent être différentes. Cette procédure a pour but d'équilibrer les tailles des vues partielles ainsi que de mélanger les adresses logiques à l'intérieur de celles-ci. Le nombre d'arcs reste constant durant cette procédure.

Les deux nœuds impliqués dans le mélange s'envoient la moitié de leur vue partielle. Après l'intégration de ces nouvelles références, la taille de leur vue partielle tend vers la moyenne et la somme globale en demeure inchangée. Dans ce but, les vues partielles sont des multiensembles. Ainsi, si un nœud reçoit un arc déjà connu, il le conserve en tant que doublon. De cette façon, le nombre d'arcs reste globalement constant.

Si les doublons ont un impact négatif sur les propriétés du réseau, la plupart disparaissent après la procédure de mélange. La proportion de doublons devient négligeable dès lors que le réseau grandit.

Algorithme 4 Procédure périodique de mélange de SPRAY.

```
1 : ACTIVE THREAD :
2 :     function LOOP( )                                          ▷ Tous les Δ t
3 :         P ← INCREMENTAGE(P) ;
4 :         let ⟨q, age⟩ ← GETOLDEST(P) ;
5 :         let sample ← GETSAMPLE(P \ {⟨q, age⟩} , ⌈|P|/2⌉ − 1) ⊎ {⟨p, 0⟩} ;
6 :         sample ← REPLACE(sample, q, p) ;
7 :         SENDTO(q, 'exchange', sample) ;
8 :         let sample′ ← RECEIVEFROM(q) ;
9 :         sample ← REPLACE(sample, p, q) ;
10 :        P ← (P \ sample) ⊎ sample′ ;
11 :    end function
12 :
13 : PASSIVE THREAD :
14 :    function ONEXCHANGE(o, sample)                            ▷ o : origine
15 :        let sample′ ← GETSAMPLE(P, ⌈|P|/2⌉) ;
16 :        sample′ ← REPLACE(sample′, o, p);
17 :        SENDTO(o, sample′) ;
18 :        sample′ ← REPLACE(sample′, p, o) ;
19 :        P ← (P \ sample′) ⊎ sample ;
20 :    end function
```

L'algorithme 4 montre les instructions exécutées périodiquement par chacun des nœuds. Il est divisé en deux parties : (i) le processus actif exécuté régulièrement afin d'initier un mélange de vues ; (ii) le processus passif réagissant au message du processus actif.

Les fonctions qui ne sont pas explicitement définies dans l'algorithme sont les suivantes :

- INCREMENTAGE($view$) incrémente l'âge des éléments présent dans la vue $view$ et retourne la vue modifiée ;
- GETOLDEST($view$) retourne le plus vieux des voisins présent dans la vue partielle $view$;
- GETSAMPLE($view$, $size$) retourne un échantillon aléatoire de la vue contenant $size$ éléments ;
- REPLACE($view$, old, new) remplace les occurrences de old par new dans la vue $view$. Si les doublons sont admis dans la vue partielle, les autoréférences restent hautement indésirables ;
- RAND() retourne un nombre flottant aléatoire compris entre 0 et 1.

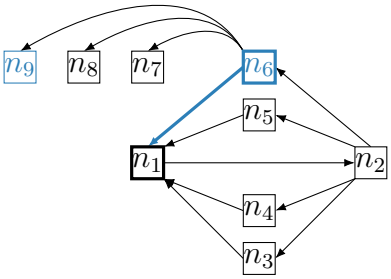
Dans le processus actif, la fonction LOOP est appelée tous les intervalles Δ de temps. La fonction incrémente l'âge de chacun des arcs présents dans la vue partielle P. Ensuite, le nœud le plus âgé q est choisi afin d'initier un mélange. Si le nœud q ne peut être joint alors le nœud p exécute une fonction appropriée (cf. §3.3.3) jusqu'à trouver un nœud avec lequel communiquer. Ainsi, le « vieillissement » permet d'accélérer la suppression des références obsolètes. Une fois que le nœud initiateur p a trouvé un nœud avec qui effectuer l'échange, il sélectionne un échantillon de sa vue partielle tout en excluant la référence à q et en s'incluant lui-même. La taille de l'échantillon correspond à la moitié de la taille de la vue partielle avec au minimum 1 nœud : sa propre référence (cf. ligne 5).

Lorsque le nœud q reçoit la demande d'échange, l'événement ONEXCHANGE est déclenché et q renvoie la moitié de sa vue partielle.

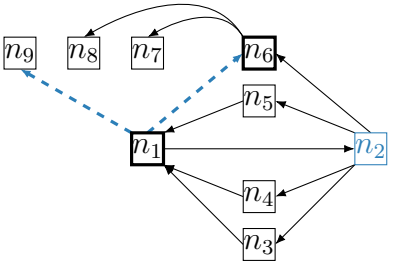
Puisqu'il existe potentiellement des doublons dans les vues partielles, les nœuds pourraient envoyer des références de leur vis-à-vis par mégarde ce qui créerait des boucles (p envoie des références de q à q).

Les lignes 6 et 16 remplacent les autoréférences (où p envoie des références de q à q) dans l'échantillon dues à la présence de doublons dans les vues. Après réception des échantillons respectifs, les nœuds suppriment l'échantillon qu'ils ont envoyé avant d'intégrer celui qu'ils ont reçu.

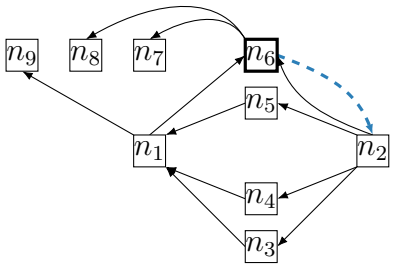
La figure 3.9 décrit le mécanisme périodique de SPRAY. Ce scénario suit celui de la figure 3.8 : le nœud n_1 vient de rejoindre le réseau. Le nœud n_6 initie un échange avec n_1



(a) Le nœud n_6 initie un mélange avec n_1 lui envoyant $\{n_6, n_9\}$.



(b) Le nœud n_1 reçoit le message de n_6. En réponse, il envoie le multiensemble $\{n_2\}$. Il ajoute $\{n_6, n_9\}$ à sa vue.



(c) Le nœud n_6 reçoit la réponse de n_1 et ajoute $\{n_2\}$ à sa vue partielle.

FIGURE 3.9 – Exemple d'exécution de la procédure de mélange dans SPRAY.

qui est ici le plus vieux nœud de sa vue partielle. n_6 choisit $\lceil |P_6| \div 2 \rceil - 1 = 1$ nœud aléatoirement parmi son voisinage. Dans ce cas, il choisit n_9 parmi $\{n_9, n_8, n_7\}$. Il envoie à n_1 cet échantillon auquel est ajoutée sa propre identité. En réponse, le nœud n_1 choisit $\lceil |P_1| \div 2 \rceil = 1$ nœud de sa vue partielle. L'échantillon composé du seul voisin n_2 est envoyé. Immédiatement après, n_1 peut supprimer l'échantillon envoyé et intégrer celui reçu résultant en une vue partielle composée de $\{n_6, n_9\}$. De la même façon, après réception de l'échantillon de n_1, n_6 supprime et intègre les échantillons adéquats, résultant en une vue partielle composée de $\{n_2, n_7, n_8\}$.

La procédure d'échange tend à réduire l'écart de taille des vues partielles. De plus, elle disperse les références des nœuds afin de supprimer les groupes trop denses dus à la procédure d'entrée dans le réseau.

En ce qui concerne le temps de convergence de l'algorithme de mélange, il existe une relation proche entre Spray et le protocole d'agrégation proactif présenté dans [48, 71]. Celui-ci déclare que, sous l'hypothèse d'un échantillonnage suffisamment aléatoire, la valeur moyenne μ et la variance σ^2 a un cycle i sont :

$$\mu_i = \frac{1}{|\mathcal{N}|} \sum_{x \in \mathcal{N}} a_{i,x} \qquad \sigma_i^2 = \frac{1}{|\mathcal{N}|-1} \sum_{x \in \mathcal{N}} (a_{i,x} - \mu_i)^2$$

avec $a_{i,x}$ la valeur stockée par le nœud n_x au cycle i. La variance estimée doit converger vers 0 au cours des cycles. En d'autres termes, les valeurs se rapprochent les unes des autres au cours des cycles. Dans le cas de Spray, cette valeur $a_{i,x}$ est la taille de la vue partielle du nœud n_x au cycle i. En effet, chaque échange entre les nœuds n_1 et n_2 entraîne une agrégation dont le résultat est : $|P_1| \approx |P_2| \approx (|P_1| + |P_2|) \div 2$. De plus, à chaque cycle, chaque nœud est impliqué dans un protocole d'échange au moins une fois (celui qu'il initie), et dans le meilleur des cas $1 + Poisson(1)$ (celui qu'il initie et, en moyenne, celui qu'il reçoit d'un autre nœud). Cette relation étant établie, nous en déduisons que la taille des vues partielles des nœuds de Spray converge en temps exponentiel vers la moyenne globale. De plus, chaque cycle réduit la variance du système à un taux compris entre $1 \div 2$ et $1 \div (2\sqrt{\mathrm{e}})$.

3.3.3 Quitter le réseau

Les nœuds quittent le réseau sans notification. Ainsi, les départs et défaillances sont gérés de façon identique. Sans réaction appropriée, la suppression d'arcs entraînerait l'effondrement du réseau.

Lorsqu'un nœud rejoint un réseau, il y injecte $1 + \ln(|\mathcal{N}|)$ arcs. Néanmoins, après plusieurs échanges de voisinage, sa vue partielle se trouve remplie d'autres références. Ainsi, lorsqu'il quitte le réseau, il entraîne la suppression de $\ln(|\mathcal{N}|)$ arcs provenant de sa vue partielle, et $\ln(|\mathcal{N}|)$ arcs provenant des nœuds l'ayant dans leur vue partielle. Par conséquent, $2\ln(\mathcal{N}|)$ arcs sont supprimés au lieu des $1 + \ln(|\mathcal{N}|)$ arcs. Afin d'y remédier, chaque nœud qui détecte une défaillance ou un départ peut rétablir une connexion avec un de ces voisins

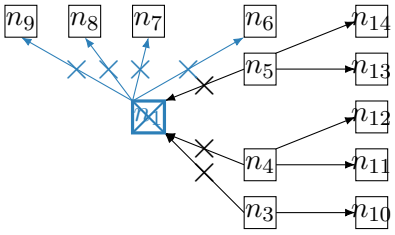
en introduisant un doublon – doublon qui disparaîtra rapidement après quelques mélanges. La probabilité de créer un tel doublon est $1-1 \div |\mathcal{P}|$. Puisque $|\mathcal{P}| \approx \ln(|\mathcal{N}|)$ nœuds avaient le nœud défaillant/parti dans leur vue partielle, il est probable qu'ils recréent tous un arc, à l'exception d'un nœud. De ce fait, lorsqu'un nœud quitte le réseau, il entraîne la suppression d'un nombre d'arcs correspondant approximativement au nombre d'arcs ajoutés lors de la dernière entrée.

Algorithme 5 Gestion des défaillances et des départs de SPRAY.

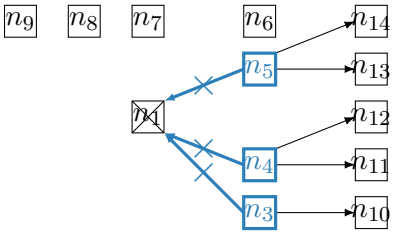
```
1 : function onPeerDown(q)                          ▷ q : nœud parti ou défaillant
2 :     let occ ← 0 ;
3 :     for each ⟨n, age⟩ ∈ P do                    ▷ supprime et compte
4 :         if (n = q) then
5 :             P ← P \ {⟨n, age⟩} ;
6 :             occ ← occ + 1 ;
7 :         end if
8 :     end for
9 :     for i ← 0 to occ do                         ▷ Duplique de manière probabiliste
10 :        if (rand() > 1/(|P|+occ)) then
11 :            let ⟨n, _⟩ ← P[⌊rand() * |P|⌋] ;
12 :            P ← P ⊎ {⟨n, 0⟩} ;
13 :        end if
14 :    end for
15 : end function

16 : function onArcDown(q, age)                     ▷ q : nœud d'arrivée dont l'arc est défaillant
17 :    P ← P \ {⟨q, age⟩} ;
18 :    let ⟨n, _⟩ ← P[⌊rand() * |P|⌋] ;
19 :    P ← P ⊎ {⟨n, 0⟩} ;                          ▷ Duplique systématiquement
20 : end function
```

L'algorithme 5 montre la manière dont SPRAY gère les départs et les défaillances. La fonction onPeerDown définit la réaction de SPRAY lorsque un nœud q est détecté comme parti ou défaillant. Dans un premier temps, la fonction compte les occurrences du nœud dans la vue partielle et en supprime les arcs. Dans un second temps, une boucle ajoute de manière probabiliste des doublons à des nœuds déjà connus. La probabilité dépend de la taille de la vue partielle avant suppression.

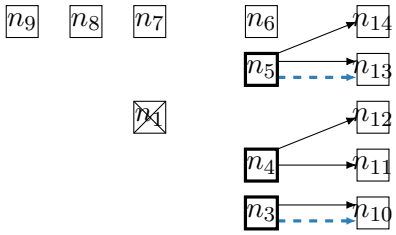
La figure 3.10 montre un exemple de réaction lors du départ d'un nœud n_1. Ce dernier emporte avec lui 7 arcs rendus inutilisables. Les nœuds n_3, n_4, et n_5 ont toujours une référence obsolète vers n_1 dans leur vue partielle. Le nœud n_5 a $1 - 1 \div |P_5| = 2 \div 3$ chances de remplacer l'arc. Dans cet exemple, il double sa référence à n_{13}. De la même manière, n_3 et n_4 ne parviennent plus à communiquer avec n_1 et agissent en conséquence. Seul n_3 crée un doublon remplaçant. Au total, 5 arcs ont été supprimés.



(a) Le nœud n_1 quitte le réseau.



(b) Les nœuds n_{3-5} s'aperçoivent qu'ils ne peuvent plus communiquer avec n_1



(c) Les nœuds n_3 et n_5 établissent des arcs doublons dans leur vue partielle.

FIGURE 3.10 – Exemple de gestion des départs et des défaillances dans SPRAY.

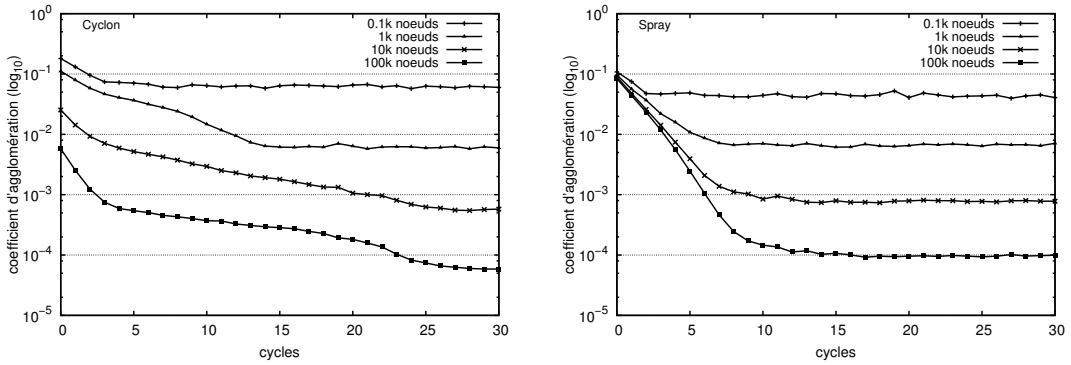
La figure 3.10 montre également que la connexité n'est pas garantie lors du départ d'un nœud. En effet, si le nœud n_1 est le seul pont reliant deux groupes de nœuds, l'ajout d'arcs est insuffisant.

L'algorithme 5 distingue les nœuds injoignables des arcs inutilisables. En effet, la fonction ONARCDOWN gère les connexions dont la création a échoué. Ces arcs sont systématiquement remplacés par un doublon. De ce fait, le nombre d'arcs reste bien constant. La distinction ONPEERDOWN avec ONARCDOWN est nécessaire car la première fonction doit supprimer un petit nombre d'arcs. Sans cette suppression, le nombre d'arcs augmenterait de manière incontrôlée au gré des entrées et sorties de nœuds.

3.4 Propriétés

SPRAY est un protocole d'échantillonnage aléatoire de pairs adaptatif. Cette section a pour objectif d'examiner ses propriétés empiriquement au travers d'une série de simulations. Les métriques auxquelles nous nous intéressons sont communément employées dans l'étude de performance de ce type de protocole ou en étude des graphes. Celles-ci incluent le coefficient d'agglomération (*clustering coefficient*), la taille moyenne du plus court chemin, la distribution des arcs entrants, l'évolution du nombre d'arcs, les composantes connexes et la robustesse. À cela nous ajoutons une analyse sur les doublons d'arcs, spécificité de SPRAY.

Les expérimentations ont été effectuées sur le simulateur PEERSIM [70]. Le code relatif



(a) Coefficient d'agglomération de CYCLON (b) Coefficient d'agglomération de SPRAY

FIGURE 3.11 – L'axe des abscisses marque le temps écoulé en nombre de cycles. L'axe des ordonnées note le coefficient d'agglomération sur une échelle logarithmique de base 10.

aux protocoles d'échantillonnage CYCLON, SCAMP, et SPRAY sont disponibles sur la plateforme Github à l'adresse : `http://github.com/justayak/peersim-spray`.

3.4.1 Coefficient d'agglomération

Le coefficient d'agglomération désigne une mesure de groupement des nœuds. En d'autres termes, si un petit groupe possède un grand nombre d'arcs les reliant, alors la mesure grimpe. Ainsi, les graphes complets affichent un coefficient maximal. À l'inverse, les graphes aléatoires affichent un coefficient très bas : la répartition des arcs ainsi que leur nombre ne favorise pas l'apparition de groupes.

Plusieurs formules existent pour calculer ce coefficient. Nous employons le coefficient local moyen $\overline{C}$ [102] effectuant la moyenne du coefficient C_x de chaque nœud n_x :

$$\overline{C} = \frac{1}{|\mathcal{N}|} \sum_{x \in \mathcal{N}} C_x \qquad \text{avec } C_x = \frac{|\{(n_y, n_z) \in P_x, n_y \in P_z \vee n_z \in P_y\}|}{|P_x|.(|P_x|-1)}$$

Objectif : Montrer que SPRAY converge rapidement vers une topologie ne comportant pas de groupes fortement connexes.

Description : Les simulations prennent pour cible des réseaux composés respectivement de 0.1k, 1k, 10k, et 100k nœuds. Le représentant des approches à taille fixe est CYCLON. Ce dernier est configuré de manière optimale pour 1k nœuds ($\ln(1000) \approx 7$ voisins). De plus, les nœuds utilisant CYCLON échangent 3 de leurs 7 voisins à chaque mélange.

Résultat : La figure 3.11 montre que CYCLON démarre avec un coefficient d'agglomération plus bas que SPRAY. Malgré cela, SPRAY parvient à converger plus rapidement que

Cyclon. De plus, quand le nombre de nœuds augmente dans le réseau, le temps de convergence de Cyclon en souffre fortement. À l'opposé, Spray converge très rapidement quel que soit la taille du réseau. La figure 3.11 montre aussi que les deux approches convergent vers un petit coefficient caractéristique des graphes aléatoires. Néanmoins, Cyclon et Spray n'atteignent pas les mêmes valeurs après convergence. À l'exception du cas où Cyclon est configuré de manière optimale, les valeurs obtenues par Spray sont soit au dessous – lorsque les vues de Cyclon sont trop grandes – ou au dessus – lorsque les vues de Cyclon sont trop petites. Globalement, Spray (i) converge vers un coefficient d'agglomération stable (ii) reflétant les besoins dus à la taille du réseau.

Explication : Cyclon commence avec un coefficient d'agglomération plus faible. En effet, lorsqu'un nœud rejoint le réseau, il est annoncé au reste du réseau via une dissémination aléatoire. Ainsi, le réseau de départ est déjà légèrement équilibré au moment où la simulation commence. En ce qui concerne Spray, un nouvel arrivant n'annonce son entrée qu'au voisinage de son contact. De ce fait, le réseau est fortement déséquilibré au départ de l'expérience quelle que soit la taille du réseau. Malgré cela, Cyclon ne converge pas aussi vite que Spray vers un coefficient stable. En effet, la taille fixe de sa vue partielle ainsi que le nombre d'entrées de chaque mélange contraint la qualité des échanges. Le coefficient d'agglomération mesure la connexité du voisinage de chaque nœud. Plus particulièrement, cela mesure à quel point le réseau est proche d'un graphe complet. Cela dépend donc de la taille des vues partielles qui, pour Cyclon, est fixée à la configuration. Ainsi lorsque le nombre de nœuds est multiplié par 10, le coefficient s'en trouve divisé par 10. En revanche, les nœuds utilisant Spray ont une taille de vue partielle reflétant automatiquement la taille du réseau. Ainsi, quand le réseau contient 1k nœuds, les vues partielles s'adaptent à cette taille. Par conséquent, Spray est très légèrement en dessous de Cyclon dans ce scénario. La taille moyenne des vues partielles est de 7.4 pour ce premier contre 7 pour ce second. En étendant ce raisonnement aux autres tailles de réseau, Spray converge vers une valeur plus basse lorsque les vues partielles de Cyclon sont trop grandes (0.1k nœuds), et vers une valeur plus haute lorsque les vues partielles de Cyclon sont trop petites (10k et 100k nœuds).

3.4.2 Plus court chemin moyen

La moyenne des plus courts chemins permet de mesurer à quel point les nœuds sont proches les uns des autres. Plus précisément, il s'agit de compter le nombre minimum d'arcs intermédiaires entre deux nœuds distincts. Ainsi, même s'il existe plusieurs chemins pour aller d'un nœud n_i à un nœud n_j, seul nous importe le plus court. Les graphes aléatoires possèdent un plus court chemin moyen très petit grandissant en $\frac{\ln |\mathcal{N}|}{\ln \ln |\mathcal{N}|}$ lorsque le nombre d'arcs suit $|\mathcal{N}| \ln |\mathcal{N}|$.

Cette métrique permet de cerner l'efficacité de la diffusion d'information dans le réseau. Par exemple, si le plus court chemin d'un nœud n_i à un nœud n_j est de 3, cela signifie qu'un

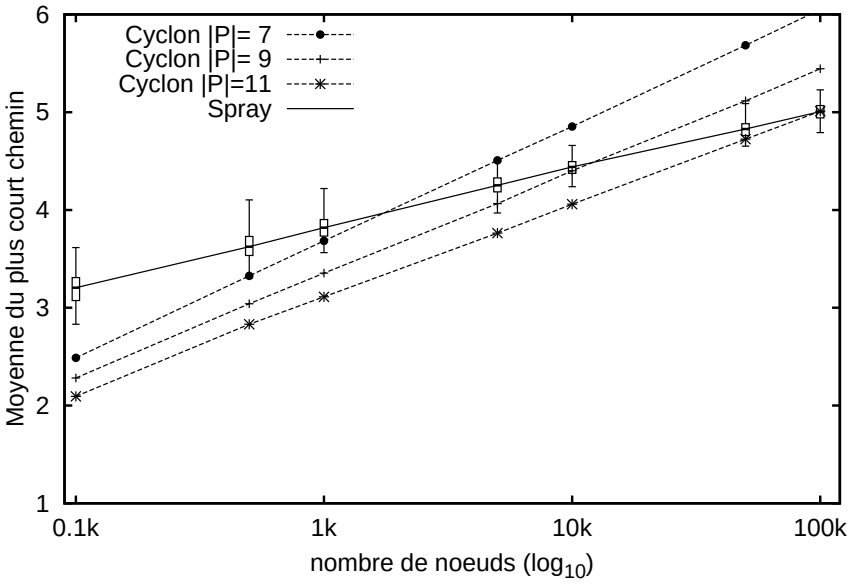


FIGURE 3.12 – La taille moyenne des plus courts chemins dans SPRAY et CYCLON. L'axe des abscisses montre le nombre de nœuds appartenant aux réseaux sur une échelle logarithmique de base 10. L'axe des ordonnées montre la taille moyenne des plus courts chemins.

message émanant de n_i parviendra à n_j par seulement 2 nœuds intermédiaires.

Objectif : Montrer que le caractère adaptatif de SPRAY permet un meilleur passage à l'échelle du plus court chemin moyen.

Description : Les mesures sont effectuées sur un sous-ensemble de nœuds choisis aléatoirement parmi les membres du réseau. Le procédé est répété 100 fois afin d'éviter tout effet de bord dû à l'indéterminisme des protocoles d'échantillonnage. Les expérimentations concernent CYCLON configuré de façon optimale pour différentes tailles de réseau. CYCLON avec une taille de vue partielle de 7 cible environ 1.1k nœuds. CYCLON avec une taille de vue partielle de 9 cible environ 8.1k nœuds. CYCLON avec une taille de vue partielle de 11 cible environ 60k nœuds. Lors de toutes ces simulations, les mesures sont effectuées après convergence. Celles-ci sont faites lorsque la taille du réseau atteint 0.1k, 0.5k, 1k, 5k, 10k, 50k, et 100k nœuds.

Résultat : La figure 3.12 montre que CYCLON et SPRAY ont tous deux un chemin moyen relativement petit. Ainsi, les informations peuvent être disséminées à tous les membres du réseau très rapidement. La figure 3.12 montre aussi que chaque exécution de CYCLON prise séparément peut être divisée en trois phases. Tout d'abord, la phase où les vues partielles de CYCLON sont surdimensionnées. Cette configuration de CYCLON possède un plus court che-

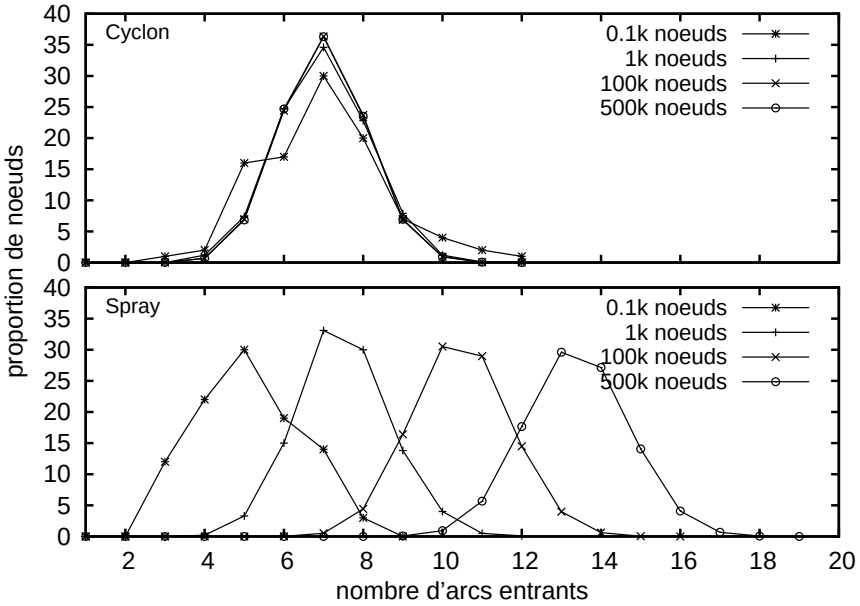


FIGURE 3.13 – Distribution du nombre d'arcs entrants à chaque nœud dans CYCLON et dans SPRAY. L'axe des ordonnées montre la proportion de nœuds ayant une vue entrante de la taille correspondante sur l'axe des abscisses. Les figures du haut et du bas sont respectivement dédiées aux exécutions concernant CYCLON et SPRAY.

min moyen inférieur à SPRAY. Lorsque les vues partielles sont optimales, CYCLON et SPRAY montrent des résultats similaires. Enfin, SPRAY montre une meilleure efficacité lorsque les vues partielles de CYCLON sont trop petites. Malgré tout, SPRAY passe mieux à l'échelle que CYCLON. En effet, le coefficient directeur de la courbe de SPRAY est inférieur au coefficient directeur de chacune des autres courbes de CYCLON.

Explication : Les mesures sont toutes effectuées après convergence, lorsque les réseaux possèdent une topologie proche des graphes aléatoires. Dans de tels graphes, la taille du plus court chemin moyen reste petite. La seconde observation concerne chacune des configurations de CYCLON comparée à SPRAY. Une surestimation de la taille du réseau pour CYCLON est meilleure en terme de connexité entre nœuds et résulte en de plus courts chemins en moyenne. En revanche, dès que la taille des vues partielles devient insuffisante pour le réseau, SPRAY devient plus efficace car il possède de plus grandes vues partielles. Puisque SPRAY suit toujours la taille optimale de la vue partielle, il passe mieux à l'échelle en terme de taille du réseau.

3.4.3 Distribution des arcs entrants

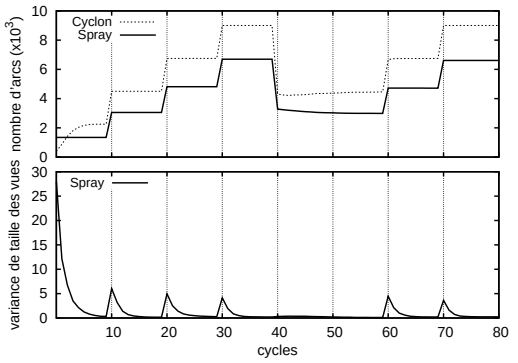
La distribution des arcs entrants correspond au nombre de fois qu'un nœud est référencé dans une vue partielle. À cet égard, la distribution entre des vues entrantes et sortantes est différente : la taille des vues partielles sortantes est contrôlée, en revanche les vues entrantes sont aléatoirement remplies par d'autres nœuds au cours de l'exécution du protocole. Certains nœuds sont donc moins représentés que d'autres. Au même titre que la distribution des vues partielles, la distribution des arcs sortants permet de déduire des faits, par exemple, sur la robustesse du réseau ou sur l'équilibre de la charge.

Objectif : Montrer que la distribution des arcs entrants dans Spray suit l'évolution du réseau.

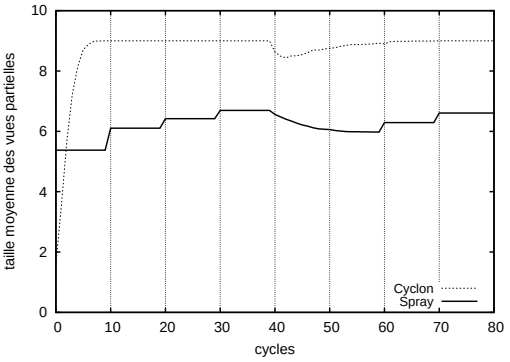
Description : Dans cette expérimentation, Cyclon est configuré avec une vue partielle contenant 7 voisins ciblant 1.1k nœuds. Les mesures sont effectuées après convergence sur des réseaux comprenant 0.1k, 1k, 100k et 500k nœuds.

Résultat : Le haut et le bas de la figure 3.13 montrent la distribution des arcs entrants pour Cyclon et Spray, respectivement. Nous observons que Cyclon possède une distribution identique quelle que soit la taille du réseau. La distribution d'un réseau de 0.1k nœuds est identique à celle de 500k nœuds, avec un fort pic sur la valeur moyenne de 7 voisins. À l'opposé, la distribution des connexions entrantes de Spray suit la taille du réseau. La figure 3.13 montre que les nœuds sont très concentrés autour des valeurs moyennes. Par exemple, lors de l'exécution du protocole Spray avec 500k nœuds, la moyenne est de 13.37 avec 88% des nœuds compris entre 12 et 14 voisins inclus. Entre autre, cela signifie que la charge est équilibrée parmi les nœuds. Puisque chaque nœud est aussi important que son voisin en terme de connexité, le réseau est robuste aux défaillances.

Explication : Une fois configuré, Cyclon doit pouvoir gérer n'importe quelle taille de réseau avec une vue partielle dont la taille est constante. Proportionnellement, le nombre de fois qu'un nœud est référencé dans les vues partielles ne change pas comparé à la taille réseau. En effet, le nombre d'arcs apportés par un nœud lorsqu'il se connecte au réseau constitue autant d'arcs le ciblant. Puisque la taille de la vue partielle est constante, le nombre d'arcs entrants reste stable. En revanche, dans Spray, chaque nœud rejoignant le réseau augmente le nombre d'arcs dans le réseau. Ainsi, le nombre d'arcs entrants de chaque nœud grossit reflétant l'augmentation de la taille du réseau. La distribution de Spray se décale lentement vers de plus hautes valeurs lorsque la taille du réseau augmente. Spray ne possède pas de pics sur les valeurs moyennes car celles-ci tombent entre deux entiers. Par exemple, si la taille moyenne des vues partielles est 6.5, cela signifie que la moitié de vues ont une taille de 6, et l'autre moitié une taille de 7. De tels réseaux sont robustes aux défaillances car aucun nœud n'est plus important que son voisin en terme de connexité. Si un nœud particulier quitte le réseau ou tombe en panne, les nœuds le référençant possèdent d'autres



(a) L'axe des abscisses montre le temps écoulé en terme de cycles. L'axe des ordonnées de la figure du haut montre le nombre total d'arcs dans le réseau tandis que l'axe des ordonnées de la figure du bas montre la variance σ^2 de la taille des vues partielles.



(b) L'axe des abscisses montre le temps écoulé en terme de cycles. L'axe des ordonnées montre la taille moyenne des vues partielles.

FIGURE 3.14 – Évolution du nombre d'arcs dans un réseau dynamique : 250 nœuds rejoignent le réseau aux cycles 0, 10, 20, et 30. Puis 500 nœuds quittent le réseau au cycle 40. Enfin, 250 nœuds rejoignent le réseau au cycles 60 and 70. Au plus haut, le réseau comprend 1k nœuds.

voisins avec qui communiquer.

3.4.4 Évolution du nombre d'arcs

Un réseau est dynamique lorsque des nœuds peuvent le rejoindre et le quitter. Le protocole d'échantillonnage doit être capable de gérer efficacement de tels changements. La variance d'un ensemble de valeurs permet de quantifier les différences qui existent entre les valeurs. Appliquée aux tailles de vues partielles, cette mesure permet d'observer l'évolution du système supposé convergeant. Une variance de 0 signifie que toutes les valeurs de l'ensemble sont identiques.

Objectif : Montrer l'évolution logarithmique du nombre d'arcs des vues partielles dans un réseau dont les nœuds entrent et sortent du réseau au cours du temps. Montrer qu'à chaque changement du réseau, le système converge en temps exponentiel vers des vues partielles dont la taille est proche.

Description : La configuration de CYCLON cible un réseau de taille 8.1k nœuds, soit des vues partielles configurées à 9 voisins. Sachant qu'au plus haut la simulation comprend 1k nœuds, cela signifie que la vue partielle a été volontairement surestimée. Lors de la première moitié de la simulation, 4 groupes successifs comportant 250 nœuds chacun rejoignent le réseau. L'intervalle de temps entre ces entrées est de 10 cycles. Ainsi, le réseau

initialement vide contient 1k nœuds au bout de 40 cycles. Ensuite, la moitié des membres du réseau le quitte, soit un départ de 500 nœuds. Enfin, 2 groupes de 250 nœuds rejoignent le réseau à nouveau faisant passer la taille de celui-ci à 1k nœuds. Les mesures concernent (i) le nombre d'arcs dans le réseau au cours du temps, (ii) la variance de la taille des vues partielles au cours du temps, (iii) la taille moyenne des vues partielles au cours du temps.

Résultat : La figure 3.14 montre les résultats de cette expérimentation. La partie haute de la figure 3.14a montre le nombre d'arcs créés dans le réseau sur une échell $\times 10^3$ tandis que la partie basse de la figure montre la variance de la taille des vues partielles de SPRAY. Dans le cas des deux protocoles d'échantillonnage, nous observons qu'à chaque groupe rejoignant le réseau, le nombre d'arcs augmente. Toutefois, la surestimation de CYCLON place son nombre d'arcs bien au dessus de SPRAY (entre 1k et 2.5k arcs supplémentaires). De plus, la figure 3.14b montre que les vues partielles de CYCLON restent de taille constante pendant toute la simulation excepté lors des départs du $40^{\text{ème}}$ cycle où les vues sont progressivement purgées des arcs obsolètes. SPRAY hérite de ce dernier trait. En revanche, les vues partielles s'adaptent automatiquement aux changements du réseau. Cette progression est logarithmique et lorsque le réseau comprend 1k nœuds, les vues partielles de SPRAY ont pour taille moyenne 6.6 voisins ($\ln(1000) \approx 6.9$). La partie basse de la figure 3.14b montre que les vues partielles s'équilibrent extrêmement rapidement. Notamment les premiers cycles réduisent considérablement l'écart entre vues partielles. La figure montre aussi que les départs ne déséquilibrent pas les vues partielles.

Explication : Puisque les vues partielles de SPRAY s'adaptent à la taille du réseau, le nombre d'arcs augmente lorsque des membres s'ajoutent au réseau. Les pics de variance correspondent à la partie du protocole où les nœuds rejoignent le réseau. Les différences de hauteur des pics proviennent du fait que les nouveaux arrivants commencent avec une petite vue partielle. Les nœuds étant présents avant que le nouveau groupe ne rejoigne le réseau ont eu quelques cycles d'échange afin d'équilibrer leur vue. Par conséquent, le poids des nouveaux arrivants est proportionnellement plus faible. Le départ des 500 nœuds au cycle 40 ne perturbe pas la variance car ils sont choisis aléatoirement. Ainsi, aucun nœud ne souffre particulièrement des départs par rapport aux autres nœuds. Toutefois, nous observons une très faible décroissance du nombre d'arcs. En effet, CYCLON et SPRAY utilisent tout deux un âge relatif à l'insertion d'un arc dans sa vue partielle. À chaque mélange, le plus vieil arc est examiné. S'il est obsolète, il est retiré. Par conséquent, ce mécanisme de détection prend plusieurs cycles à détecter les départs et défaillances. Plus la vue partielle est petite, plus le système est réactif vis-à-vis des départs et des défaillances.

3.4.5 Robustesse

La robustesse d'un réseau est sa capacité à rester connexe lorsque des défaillances apparaissent. Ici, nous ne prenons en compte que les défaillances aléatoires non-corrélées. Lors-

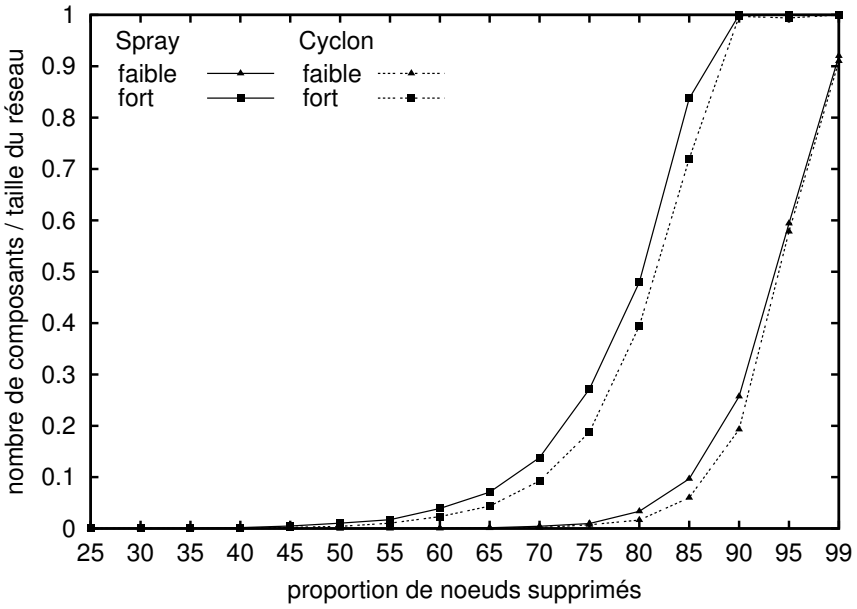


FIGURE 3.15 – Robustesse de CYCLON et SPRAY face aux défaillances aléatoires massives. L'axe des abscisses montre le pourcentage de nœuds supprimés en une fois. L'axe des ordonnées montre le ratio entre le nombre de composantes connexes et la taille du réseau après suppression.

qu'un nœud quitte le réseau, qu'il soit défaillant ou non, des arcs deviennent obsolètes. Si ces arcs constituaient les seuls liens entre deux parties du réseau, alors apparaissent des partitions. Dans les graphes aléatoires, un grand nombre de nœuds doivent disparaître avant que de telles partitions apparaissent.

Compter le nombre de composantes fortement connexes dans un réseau permet d'estimer la surface atteignable par les protocoles de dissémination d'information. Par exemple, il y a deux composantes fortement connexes dans un réseau dont une partie peut atteindre l'autre sans que l'inverse soit possible.

Compter le nombre de composantes faiblement connexes dans une réseau permet d'estimer le moment où celui-ci n'est plus réparable. Un réseau est réparables si après quelques échanges de vues partielles, les informations peuvent être disséminées à tous les participants encore présents dans le réseau. Il suffit d'un arc entre deux composantes fortement connexes pour que le réseau soit réparable. Sans ce lien, les composantes sont faiblement connexes.

Objectif : Montrer que Spray est robuste face aux défaillances aléatoires massives.

Description : Cyclon est configuré de façon à fournir des vues partielles contenant 9 voisins. Le réseau compte 10k membres. Les suppressions sont effectuées après convergence. Les nœuds sont supprimés aléatoirement et d'un trait. Allant de 25% à 95% par paliers de 5%, cela représente 16 exécutions par approche. La dernière mesure est effectuée à 99% de suppressions. Les mesures des composantes connexes sont effectuées immédiatement après suppression.

Résultat : La figure 3.15 montre le ratio de composantes fortement et faiblement connexes du réseau après suppression massive de nœuds. Tout d'abord, la figure montre que les deux protocoles d'échantillonnage de pairs, Spray et Cyclon, ne souffrent de partitionnement qu'après un fort taux de suppressions, Cyclon étant légèrement meilleur dans ce cas. La figure 3.15 montre que les composantes fortement connexes commencent à se dégrader lentement dès 45%, et plus rapidement à 70%. Dans ce cas, la dissémination d'information est temporairement en danger : un message peut ne pas parvenir à l'ensemble des membres du réseau. Heureusement, la figure 3.15 montre aussi que ces approches sont capables de réparer ce partitionnement. En effet, le nombre de composantes faiblement connexes ne commence à augmenter qu'à partir de 70% de pannes, ce qui signifie que des parties du réseau sont complètement disjointes et donc, au delà de toute possibilité de réparation.

Explication : Les protocoles d'échantillonnage de pairs Cyclon et Spray affichent des résultats très similaires car Cyclon est configuré pour une taille de réseau de 10k nœuds, tandis que Spray s'ajuste automatiquement à cette taille de réseau. Par conséquent, leur nombre d'arcs est très proche. Ici, Cyclon possède légèrement plus d'arcs – car Spray possède une part d'aléatoire notamment lors du choix du contact nécessaire pour rejoindre

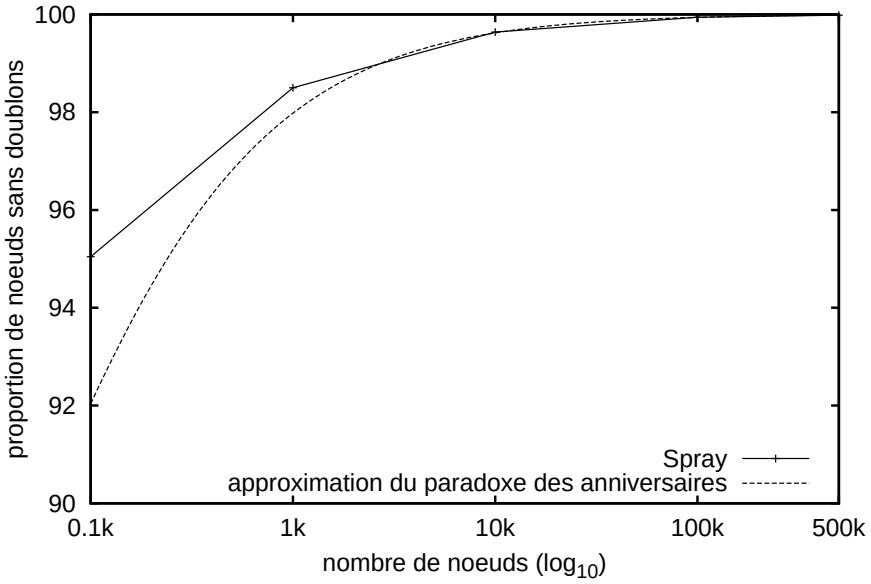


FIGURE 3.16 – Proportion de doublons parmi les nœuds. L'axe des abscisses montre la taille du réseau sur une échelle logarithmique de base 10. L'axe des ordonnées montre le pourcentage de vues ne possédant pas de doublons.

le réseau – et aucun doublon dans les vues partielles – SPRAY possède un faible taux de doublons (cf. la figure 3.16). Pour mettre en danger la dissémination d'information, il est nécessaire de supprimer un grand nombre de nœuds. En effet, chaque nœud revêt grossièrement la même importance que son voisin. Ainsi, supprimer des nœuds aléatoirement parmi ceux-ci n'affecte pas énormément le réseau dans son entièreté. Le réseau est capable de se réparer car les protocoles d'échantillonnage ne dépendent que d'arcs unidirectionnels pour fonctionner. Ainsi, s'il subsiste un arc entre deux composantes fortement connexes, les mélanges périodiques de vues partielles permettent de les raccorder.

3.4.6 Doublons

Les arcs doublons sont une spécificité de SPRAY. Ceux-ci permettent de ne pas perdre d'informations aux cours des mélanges et ainsi conserver un nombre d'arcs cohérent tout au long du cycle de vie du réseau. Toutefois, ils ont un impact négatif sur certains aspects du réseau : un arc doublon est un arc n'apportant aucun chemin supplémentaire au réseau. À cet égard, il est important que le taux de doublons reste faible.

Pour évaluer la proportion de doublons, nous en faisons l'analogie avec le paradoxe des anniversaires. Ce paradoxe déclare que le nombre de personnes à réunir pour que deux d'entre elles aient le même jour d'anniversaire est, contre toute attente, faible. À partir

de 23 personnes, la probabilité est au dessus de 50% et grimpe rapidement. Au delà des anniversaires, ce modèle permet de quantifier les collisions qu'il peut y avoir lorsque l'on tire des éléments dans un ensemble borné. Dans notre cas, il est possible d'approximer la probabilité qu'une vue n'ait pas de doublons par :

$$1 - (1 - \exp(\frac{-\ln(|\mathcal{N}|) * (\ln(|\mathcal{N}|) - 1)}{2 * |\mathcal{N}|}))$$

Objectif : Montrer que la proportion de doublons dans SPRAY devient négligeable lorsque le réseau grandit.

Description : Les mesures concernent le nombre de vues partielles contenant au moins un doublon après convergence. Celles-ci sont effectuées sur des réseaux comprenant 0.1k, 1k, 10k, 100k, et 500k nœuds.

Résultat : La figure 3.16 montre la proportion de nœuds ne possédant pas de doublons dans leur vue partielle. Nous observons qu'il existe toujours au moins une vue partielle contenant un doublon. La proportion est plus importante lorsqu'il s'agit de réseaux de petite taille (e.g. 5% pour 100 nœuds). Cette proportion diminue à mesure que la taille du réseau grandit (e.g. moins de 1% à partir de 10k nœuds). La figure 3.16 montre que l'approximation du paradoxe des anniversaires est proche des résultats obtenus par simulation confirmant empiriquement qu'il existe une relation entre les deux problèmes.

Explication : Lorsque le réseau grandit, les chances qu'un nœud particulier possède plusieurs arcs ciblant un autre nœud particulier deviennent très faibles. En effet, alors que le réseau grandit linéairement, l'ensemble des références détenu par un nœud grandit logarithmiquement. Contrairement au problème des anniversaires, la diminution en proportion n'est pas contre-intuitive au vu des complexités impliquées.

3.5 Cas d'utilisation : la dissémination de message

La diffusion de message d'un nœud à tous les autres est une fonctionnalité courante des réseaux (*broadcast*). La propagation épidémique (*gossip*) [13] constitue un moyen efficace de la mettre en place. Sa dénomination provient de son fonctionnement : un nœud choisit un sous-ensemble des membres du réseau et les "contamine" en leur envoyant le message ; les nœuds "infectés" en font de même ; un nœud ayant déjà envoyé le message ne le renvoie pas. Plus la taille du sous-ensemble choisi est élevée, plus la probabilité que le message parvienne à tous les membres du réseau augmente [28].

L'algorithme 6 montre les quelques instructions composant la dissémination épidémique de messages. L'épanouissement (*fanout*) désigne le nombre de voisins auxquels le message va être envoyé. La fonction GETPEERS renvoie autant de nœuds distincts. Dans

Algorithme 6 Algorithme de dissémination de messages.

```
1 : function BROADCAST(m)                          ▷ m : message à envoyer
2 :     let chosen ← getPeers(P, fanout) ;
3 :     for (n ∈ chosen) do
4 :         SENDTO(n, 'broadcast', m) ;
5 :     end for
6 : end function

7 : function ONBROADCAST(m)                        ▷ m : message reçu
8 :     if (¬ALREADYRECEIVED(m)) then
9 :         BROADCAST(m) ;
10 :    end if
11 : end function
```

le cadre d'approches fournissant une vue partielle de taille constante, cette variable d'épanouissement est elle aussi constante, et configurée à l'avance. Pour que les chemins empruntés lors de la dissémination soient eux aussi aléatoires malgré la fréquence beaucoup plus faible des échanges du protocole d'échantillonnage, la taille de la vue est nettement surestimée [34]. Par exemple, une vue partielle contient 30 arcs, mais seulement 6 sont utilisés à chaque dissémination. Dans le cadre de SPRAY, nous augmentons les tailles des vues partielles fournies : lors de l'entrée dans le réseau, un nœud n'ajoute plus qu'une seule référence au contact mais c doublons ; les vues partielles tendent vers $c \cdot \ln(|\mathcal{N}|)$ [36] ; un nœud détectant un départ ou une panne ajuste la probabilité de supprimer un arc à $c \div (|P| + occ)$. La variable d'épanouissement est configurée comme étant une fraction de la vue partielle. Ainsi l'épanouissement peut s'ajuster automatiquement pour suivre une progression logarithmique : $\ln(|\mathcal{N}|) + x$, où x est un entier positif.

La suite met en évidence l'avantage apporté par une variable d'épanouissement s'ajustant à la taille du réseau. En particulier, nous mesurerons le taux de réception totale des messages par les membres du réseau. Si au moins un membre ne reçoit pas le message, alors la réception totale a échoué. Le taux mesuré est le nombre de messages ayant été reçus par tous les membres sur le nombre de messages ayant été envoyés.

Objectif : Montrer les avantages apportés au mécanisme de dissémination par un protocole adaptatif d'échantillonnage.

Description : L'application considérée cible approximativement 100 nœuds. Les vues partielles fournies par CYCLON sont configurées pour contenir 30 voisins ($6 \cdot \ln(100) \approx 6 \cdot 5 = 30$). Nous considérons 2 configurations pour l'épanouissement : $\ln(100) + 1 \approx 6$ et $\ln(100) + 3 \approx 8$. Ces configurations garantissent une forte probabilité de réception des messages lorsque le réseau contient 100 nœuds ou moins. De même, nous configurons SPRAY pour qu'il fournisse des vues partielles 6 fois plus peuplés qu'à la normale. La va-

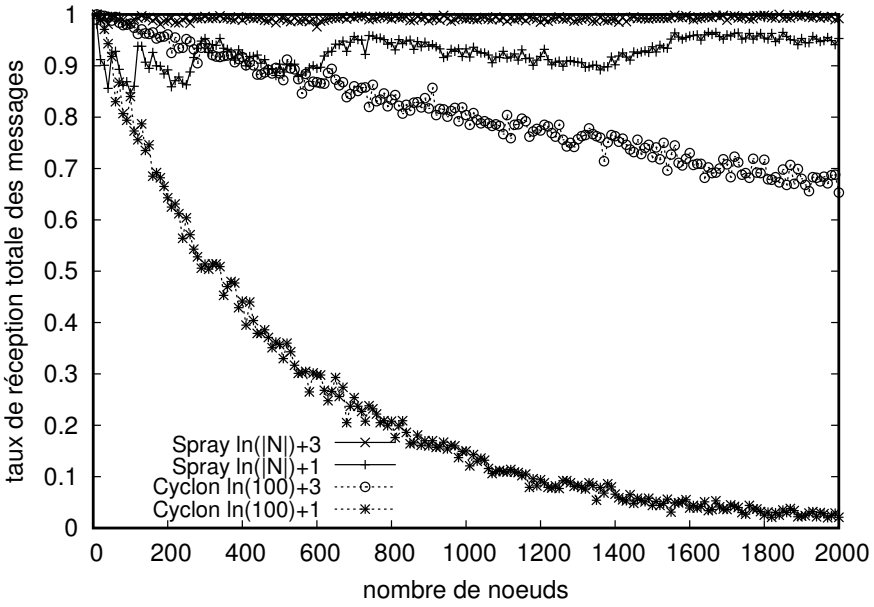


FIGURE 3.17 – Taux de réception des messages en fonction de la taille du réseau. L'axe des abscisses montre la taille du réseau en nombre de nœuds. L'axe des ordonnées montre le taux de réception total mesuré sur 1k messages.

riable d'épanouissement est configurée pour choisir 1 sixième de la vue, soit $\ln(|\mathcal{N}|) + 1$ et $\ln(|\mathcal{N}|) + 3$. Les deux protocoles d'échantillonnage aléatoire possèdent donc les mêmes configurations pour un réseau de 100 nœuds. La taille des réseaux considérés grandit de 0.1k à 2k nœuds. Nous mesurons à chaque cycle la fraction, sur 1k messages, parvenant à l'intégralité des membres du réseau.

Résultat : La figure 3.17 présente les résultats de cette simulation. Tout d'abord, nous observons que le taux de réception des messages avec CYCLON souffre d'une constante décroissance. De moins en moins de messages parviennent à l'ensemble des nœuds lorsque le réseau s'agrandit. La configuration dont la variable d'épanouissement est la plus élevée fournie de meilleurs résultats. En revanche, le taux de réception des messages de SPRAY reste stable malgré la présence de petits sauts dans les valeurs mesurées. Un épanouissement initialisé à $\ln(|\mathcal{N}|) + 1$ conduit à un taux aux environs de 90%. Un épanouissement initialisé à $\ln(|\mathcal{N}|) + 3$ conduit à un taux proche de 100%. Globalement, le mécanisme de dissémination épidémique présenté profite bien de la nature adaptative de SPRAY. Cela lui permet de s'ajuster aux besoins d'un réseau dont la taille fluctue au cours du temps.

Explication : Le seuil précis de connexité d'un graphe aléatoire est $\Theta(|\mathcal{N}| \ln(|\mathcal{N}|))$ arcs. En ce qui concerne la dissémination de messages, avec des vues partielles suffisamment

grandes et peuplées d'arcs vers des nœuds suffisamment aléatoires, un épanouissement fixé à $\ln(|\mathcal{N}|)$ conduit à un taux de réception aux environs de 50%. À ce stade, incrémenter cette variable augmente grandement le taux de réception. Cependant, la contribution de chaque incrémentation diminue progressivement par rapport à l'incrémentation précédente. Le mécanisme de dissémination construit au dessus de Cyclon possède un épanouissement constant, configuré pour une taille spécifique du réseau. Tant que la taille du réseau est inférieure ou égale à la taille ciblée, le taux de réception demeure élevé. En revanche, une fois cette taille dépassée, le taux de réception chute très rapidement. Spray donne la possibilité au mécanisme de dissémination d'ajuster sa variable d'épanouissement à la taille du réseau. Ainsi, le taux de réception des messages reste stable, même lorsque le réseau s'agrandit. Les petits sauts mesurés sont dus au fait que les valeurs manipulées localement par chaque nœud sont des entiers. Le bond correspond donc à une valeur arrondie augmentant suffisamment pour être incrémentée.

Objectif : Montrer comment le taux de réception des messages se comporte lors d'un pic de population, i.e., une augmentation massive de la taille du réseau suivie peut après d'une diminution.

Description : Nous configurons Spray et Cyclon de la même manière que pour la précédente simulation. Pour Cyclon, les vues partielles sont initialisées à 30 voisins avec deux valeurs pour l'épanouissement : $\ln(100)+1$ et $\ln(100)+3$. Pour Spray, les vues partielles s'ajustent automatiquement à $6 \cdot \ln(|\mathcal{N}|)$ et les valeurs d'épanouissement s'ajustent automatiquement à $\ln(|\mathcal{N}|)+1$ et $\ln(|\mathcal{N}|)+3$. La simulation commence avec 0.1k nœuds. Le réseau atteint rapidement 10k nœuds pendant le pic de popularité. Enfin il redescend à 3k nœuds. Dans cette simulation, nous mesurons le taux de réception total des messages sur 0.1k messages.

Résultat : La figure 3.18 montre les résultats de cette simulation. Tout comme lors de la simulation précédente, l'utilisation de Cyclon et d'une variable d'épanouissement prédéfinie fonctionne jusqu'à ce que la taille du réseau excède la taille prévue. Pendant le pic d'entrée dans le réseau, le taux de réception chute très rapidement. À l'inverse, le mécanisme de dissémination construit avec Spray ajuste automatiquement sa variable d'épanouissement à la taille du réseau. Ainsi, le système ne souffre pas de sévère baisse du taux de réception lors du pic d'entrée. Cependant, lors du départ massif de certains membres, le taux de réception chute. Cela est particulièrement vrai pour la configuration de Spray avec un épanouissement plus modeste de $\ln(|\mathcal{N}|)+1$. Globalement, le mécanisme de dissémination se comporte mieux avec Spray et devient résilient aux fluctuations rapides de la taille des réseaux.

Explication : La variable d'épanouissement des configurations impliquant Cyclon est constante. Ainsi, lorsque la taille du réseau dépasse la taille prévue, le taux de réception chute drastiquement. À l'opposé, la variable d'épanouissement des configurations basée sur

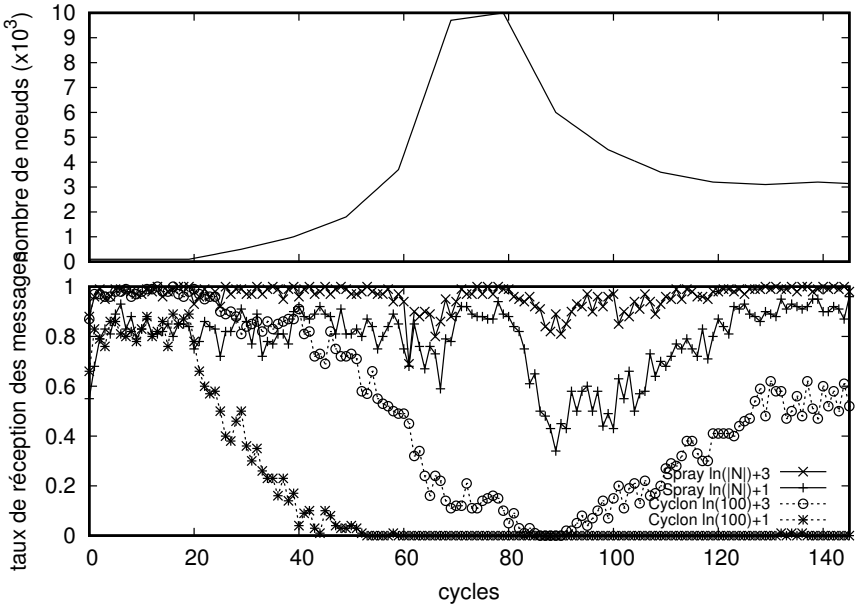


FIGURE 3.18 – Taux de réception des messages lors d'un pic d'entrée dans le réseau. L'axe des abscisses montre le temps en nombre de cycles. L'axe des ordonnées de la partie haute montre le nombre de nœuds présent dans le réseau. L'axe des ordonnées de la partie basse montre le taux de réception des messages sur 0.1k messages.

Spray suit les évolutions du réseau. Le taux de réception reste stable lors d'augmentations rapides de la taille du réseau. Cependant, à la fois Cyclon et Spray détectent les nœuds partis lors de la phase d'échange de vues partielles. Les arcs obsolètes sont malgré tout utilisés dans la dissémination. Par conséquent, un plus grand nombre de messages à tendance à se perdre. Progressivement, les arcs obsolètes disparaissent des vues partielles. Le taux de réception des messages recouvre sa valeur attendue.

3.6 Conclusion

Dans ce chapitre, nous avons présenté Spray, un protocole d'échantillonnage aléatoire de pairs s'ajustant automatiquement aux fluctuations de la taille du réseau. Chaque nœud d'un réseau de taille $|\mathcal{N}|$ se voit attribuer une vue partielle du réseau de taille $\ln |\mathcal{N}|$ avec laquelle il peut communiquer. Périodiquement, les nœuds échangent une partie de leur vue partielle avec leur voisin le plus âgé. Bien que l'identité des nœuds ne se propagent que de voisin à voisin, le réseau converge vers une topologie dont les propriétés sont stables en très peu de temps. La topologie résultant de ces mélanges de voisinages possède des propriétés proches des graphes aléatoires. Par exemple, le réseau est tolérant aux pannes et les messages se propagent efficacement.

Un développeur peut utiliser Spray afin que le trafic généré suive les variations du réseau, et ce, sans configuration préalable de sa part. Le chapitre 4 montre un exemple d'utilisation de Spray dans le contexte de l'édition collaborative temps réel dans les navigateurs Web. Ce contexte est propice à son utilisation car l'établissement de connexion d'un navigateur à l'autre s'avère coûteux. Par conséquent, conserver un nombre logarithmique de connexions s'établissant précautionneusement de proche en proche constitue un avantage. De plus, le Web favorisant les échanges et la propagation d'idées, un éditeur Web doit être capable de faire face aux pics soudains de popularité.

4

Un éditeur collaboratif temps réel dans les navigateurs

Sommaire

L'édition collaborative [25, 51] permet de répartir la rédaction d'un document au cours du temps et à travers l'espace [24, 43, 51]. Elle améliore l'implication des participants et la qualité des documents produits [39, 79]. Les éditeurs collaboratifs temps réel permettent à un utilisateur de visualiser, modifier et partager un document. Le partage du document avec un ou plusieurs collaborateurs permet à ces derniers d'avoir à leur tour accès à la lecture et la rédaction du document.

Depuis la première démonstration d'éditeur collaboratif temps réel en 1968 [27], les technologies ont bien changé. Entre autres, le Web est devenu un terrain fertile pour les nouvelles applications [57]. Ces dernières sont bien souvent préférées par les utilisateurs aux applications natives ou aux extensions pour leur facilité d'usage [68] ; elles sont bien souvent préférées par les entreprises qui s'évitent le développement d'applications spécifiques aux systèmes d'exploitation et aux matériels informatiques [68].

Les éditeurs Web ont des fins variées. Certains éditeurs Web permettent la rédaction de documents génériques [32, 40, 41, 53], tandis que d'autres se concentrent sur la rédaction de documents scientifiques [5, 61, 81, 108] ou sur l'écriture de code [58, 91].

Les éditeurs Web tels que Google Docs [40] ou Etherpad [32] ont grandement contribué à rendre ces outils populaires. L'édition collaborative est devenue aisée pour des millions d'utilisateurs à travers le monde. Un simple lien permet à plusieurs collaborateurs d'accéder, puis lire et modifier un document en temps réel. Cette facilité d'accès est rendue possible par le biais d'un serveur centralisé appartenant à un fournisseur de service hébergé sur le Nuage [66]. Cette organisation, où quelques serveurs sont en charge d'un grand nombre de clients, pose des problèmes de confidentialité, de censure, de passage à l'échelle et de point unique de défaillance. Seule la décentralisation permet de résoudre ces problèmes. Toutefois, sa possibilité d'intégration dans les navigateurs Web via WebRTC [45] n'est que récente.

Ce chapitre présente Crate, le premier[1] éditeur collaboratif temps réel complètement décentralisé et fonctionnant dans les navigateurs Web. À ce titre, un document n'appartient qu'aux membres de la session d'édition. Les collaborateurs, par leur présence, participent au bon fonctionnement de la session d'édition. Pour passager à l'échelle, Crate utilise à la fois Spray (cf. §3) pour que chaque modification soit propagée efficacement à l'ensemble des éditeurs impliqués dans la rédaction et LSeq (cf. §2) pour que les répliques du document hébergé par chaque éditeur convergent. Le trafic généré par l'application est logarithmique par rapport à la taille de la session d'édition et polylogarithmique par rapport au nombre d'insertions effectuées dans le document.

Le reste de ce chapitre s'organise de la façon suivante : la section 4.1 introduit Crate, son architecture et ses composants internes, ainsi que son fonctionnement ; la section 4.2 décrit les résultats obtenus lors d'une expérimentation à large échelle impliquant jusqu'à 600 navigateurs ; la section 4.3 conclut ce chapitre.

4.1 Crate : un éditeur décentralisé dans les navigateurs

Crate est un éditeur décentralisé permettant l'édition en temps réel de documents dans les navigateurs Web. Cette section décrit son architecture avant de détailler chacun des composants constituant cette architecture.

La figure 4.1 montre l'architecture en 4 couches de Crate. Chacune de ces couches peut devenir un obstacle au passage à l'échelle de l'éditeur :

(i) la couche de communication comprend le mécanisme d'appartenance au réseau et la propagation des messages dans ce réseau ;

[1] À notre connaissance

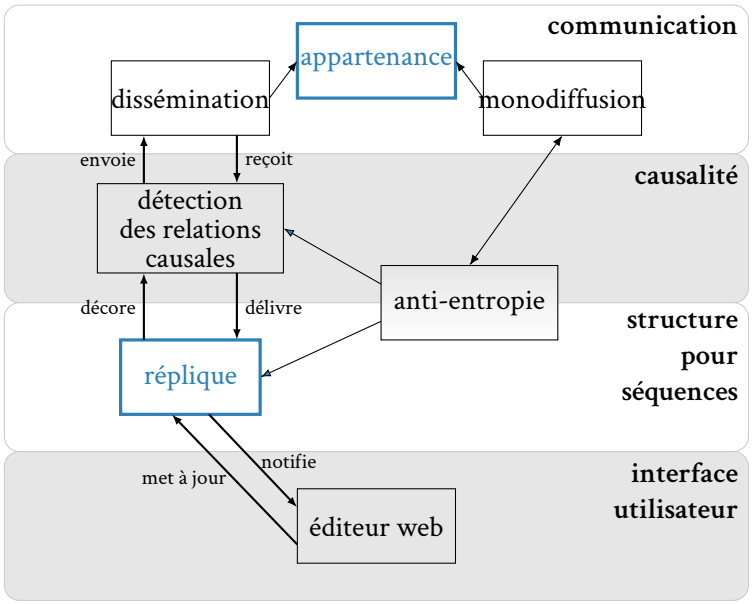


FIGURE 4.1 – Architecture en 4 couches de CRATE.

(ii) la couche de causalité comprend la structure permettant de lier les opérations entre elles afin qu'elles soient intégrées dans un ordre reflétant une forme de causalité, e.g., elle assure que les opérations de suppression suivent toujours les opérations d'insertion de l'élément correspondant ;

(iii) la couche de structure pour séquences dont les opérations d'insertion et de suppression doivent garantir des répliques convergeantes du document ;

(iv) la couche d'interaction homme-machine fournissant les outils avec lesquels l'utilisateur peut interagir.

La partie gauche de la figure montre l'enchaînement le plus courant : lorsqu'un participant effectue une opération sur le document, l'opération est appliquée à la séquence répartie. L'opération est ensuite décorée avec des métadonnées correspondant à la causalité. L'éditeur propage l'opération en utilisant le voisinage de l'éditeur fourni par le protocole d'appartenance au réseau. À l'inverse, lorsque l'éditeur reçoit une opération, il vérifie si cette dernière est prête à être intégrée. Lorsque la condition est vérifiée, l'éditeur intègre l'opération à la réplique de la séquence. L'interface utilisateur est notifiée du changement.

La partie droite de la figure correspond à la stratégie de rattrapage où un participant a peut-être manqué quelques opérations à cause de pertes de messages dans le réseau, ou simplement car il était hors-ligne pendant un moment. Ainsi, dès que l'éditeur est en ligne, il vérifie régulièrement auprès de ses voisins s'il lui manque des opérations [23, 98].

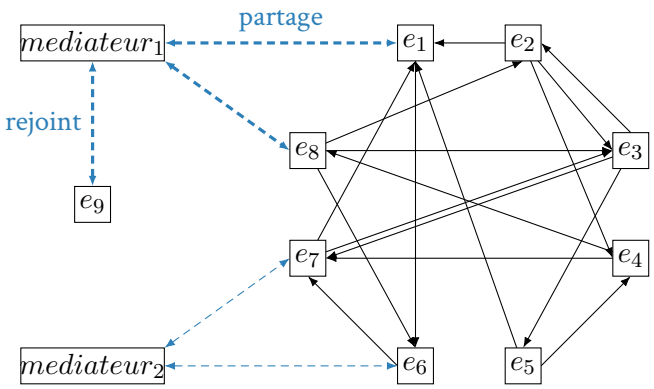


FIGURE 4.2 – Fonctionnement d'une session d'édition.

4.1.1 Communications

Les sessions d'édition peuvent rassembler de petits groupes comme de larges groupes pendant leur durée de vie. Par exemple, un cours de formation en ligne ouverte à tous (*MOOC*) peut commencer avec un grand nombre d'étudiants dont le nombre peut s'amoindrir très rapidement par de manque d'intérêt [15]. De plus, les sessions d'édition varient en taille selon la portée du document. Par exemple, un document décrivant un projet personnel et dont la visibilité est limitée aux amis rassemble significativement moins de monde qu'un document décrivant un grand événement, tel que le rapport collaboratif d'une conférence. La couche de communication doit être en mesure de gérer de manière transparente toute session d'édition, quelle que soit sa taille, tout en passant à l'échelle.

CRATE utilise SPRAY (cf. chapitre 3) afin d'ajuster automatiquement son fonctionnement à la session d'édition. Ainsi, chaque éditeur possède une vue partielle avec laquelle communiquer. La taille de cette dernière croît et décroît logarithmiquement par rapport à la taille du réseau. Si une session d'édition démarre avec 10 auteurs, ils auront 2.3 voisins en moyenne. Si la session d'édition grandit pour atteindre le millier de participants, ceux-ci auront 6.9 voisins en moyenne. Enfin, si l'édition se retrouve avec 10 participants à nouveau, ils auront à nouveau 2.3 voisins en moyenne.

La diffusion des messages fait un usage intensif des voisinages. Lorsqu'un utilisateur procède à des changements dans le document, l'éditeur l'envoie à son voisinage. Chacun des voisins ayant reçu le message le transmet à son tour à son voisinage. Les modifications du document atteignent rapidement tous les éditeurs par transitivité. La complexité en communication chez chaque éditeur est bornée par $\mathcal{O}(M \ln(R))$, où M est la taille du message et R est le nombre de répliques connectées au moment de la propagation.

La figure 4.2 décrit la procédure d'entrée dans le réseau. Une session d'édition, composée de 8 éditeurs existe. Celle-ci est inaccessible depuis l'extérieur. Pour rejoindre le réseau,

au moins un des éditeurs doit partager son accès. Pour ce faire, il crée une connexion avec un serveur de signalisation facilement accessible via *websocket*. L'éditeur ayant partagé son accès donne à son collègue un lien HTTP contenant suffisamment d'informations pour qu'il retrouve le médiateur et demande à rejoindre la session d'édition. Lorsqu'il clique sur l'adresse, une connexion est établie avec le serveur de signalisation qui va jouer le rôle de médiateur afin de créer le premier canal de communication WebRTC entre le nouveau membre et l'éditeur partageant l'accès. Une fois celui-ci établi, le nouvel arrivant se déconnecte du médiateur et applique le protocole SPRAY en utilisant le canal WebRTC. Ensuite, les éditeurs eux-mêmes deviennent des médiateurs et permettent d'établir les connexions WebRTC d'un voisinage à l'autre.

4.1.2 Détection de la causalité

Pour préserver des répliques cohérentes, le même résultat doit advenir lors de la création de l'opération et de son intégration. Souvent, le comportement d'une opération dépend d'autres opérations précédemment intégrées. Par exemple, la génération d'une suppression nécessite l'existence de l'élément ciblé, et donc, que l'opération d'insertion de cet élément soit intégrée. Ces relations de précédence (*happens before*) [56] contraignent l'ordre d'intégration des opérations. Malheureusement, l'ordre causal est très coûteux : maintenir les relations causales d'une opération par rapport à toutes les autres nécessite au minimum $\mathcal{O}(W)$ où W est le nombre de participants ayant effectué au moins une opération [18]. La réception causale introduit dans la complexité en communication un facteur linéaire sur le nombre de participants. Si cela reste acceptable pour les petits groupes, le coût devient trop élevé pour les grands groupes.

CRATE ne contraint l'ordre d'intégration que pour les paires d'opérations liées sémantiquement : la suppression d'un élément suit toujours l'insertion de ce dernier. Si l'éditeur reçoit les opérations dans le désordre, la suppression attend l'insertion correspondante. En revanche, les insertions sont immédiatement intégrées lorsqu'elles sont reçues. Pour caractériser ces relations causales, CRATE utilise un vecteur d'horloges avec exceptions [62, 73]. Ce vecteur stocke pour chaque participant (i) un entier désignant le compteur maximal connu de ce participant et (ii) une liste d'entiers désignant les exceptions, à savoir les opérations de ce participant dont l'existence est connue mais qui ne sont pas encore reçues. Tandis qu'une telle structure conserve un coût local de l'ordre du vecteur d'horloges $\mathcal{O}(W)$, le coût en communication devient constant $\mathcal{O}(1)$.

La figure 4.3 montre une session d'édition impliquant 3 participants. Les vecteurs d'horloges avec exceptions sont initialisés à vides. Le collaborateur c_1 insère deux lettres dans le document et envoie les messages correspondants. Le collaborateur c_3 reçoit rapidement les deux messages et ajoute les identifiants des opérations à la structure de causalité. Les opérations sont intégrées dans la séquence répliquée. Au même moment, le collaborateur c_2 ne reçoit quant à lui que la seconde opération. Par conséquent, il marque la première opé-

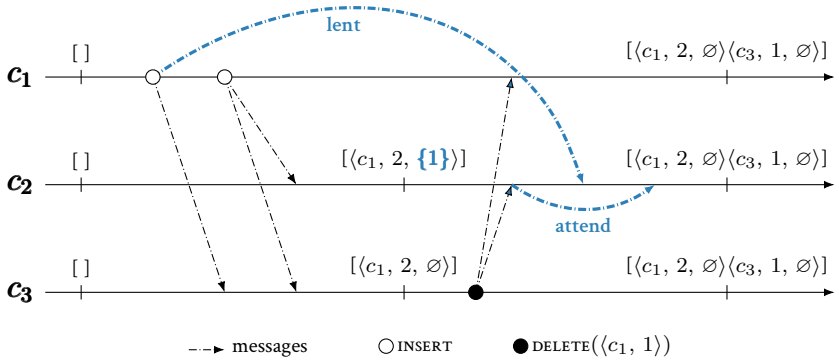


FIGURE 4.3 – Exemple de détection de relations causales. Les triples dans les vecteurs sont composés de $\langle origin, max, exceptions \rangle$.

ration de c_1 en tant qu'exception et intègre l'opération reçue. Le collaborateur c_3 supprime la première lettre insérée par c_1 et en envoie le message. Tandis que c_1 intègre la suppression immédiatement, c_2 doit attendre puisque l'opération d'insertions ciblée appartient aux exceptions. Une fois l'opération manquante reçue, l'exception disparaît et l'insertion est appliquée suivie de la suppression.

4.1.3 Anti-entropie

Lorsqu'un éditeur a besoin de récupérer l'état courant du document (e.g. lorsqu'il rejoint une session d'édition), il exécute un mécanisme d'anti-entropie. Ce mécanisme a pour objectif de détecter les divergences entre les répliques avant de les faire converger vers un état équivalent. À cela s'ajoute la contrainte de ne perdre aucune des modifications effectuées sur les répliques. Le vecteur utilisé pour détecter les relations causales sert aussi d'outil afin d'identifier les différences entre répliques. Le mécanisme d'anti-entropie [23, 98] fonctionne de la façon suivante :

(i) Un éditeur choisit un éditeur de son voisinage aléatoirement et lui envoie son vecteur d'horloges avec exceptions.

(ii) L'éditeur recevant un tel vecteur effectue la différence, entrée par entrée, avec son propre vecteur. Ces différences permettent d'identifier les opérations manquantes qui sont alors envoyées en réponse.

(iii) L'éditeur recevant la réponse intègre les opérations normalement.

CRATE utilise donc une approche basée sur les opérations lorsqu'il fonctionne normalement afin de permettre l'édition en temps réel. Lorsqu'il est en mode hors ligne ou si le

réseau subit des perturbations, alors CRATE utilise un approche basée sur les différences. Le temps de convergence des répliques est rapide grâce à la diffusion épidémique des messages et fiable grâce au mécanisme d'anti-entropie.

4.1.4 Séquence répliquée

La convergence forte à terme définit qu'un système est correct si les répliques intégrant un même ensemble d'opérations convergent vers un état équivalent [90]. La séquence répliquée a pour objectif de garantir cette propriété de convergence des copies. Ainsi, les membres d'une session d'édition lisent un document identique.

CRATE utilise un type de données sans conflits conçu pour les séquences afin de représenter ses documents. Un telle structure se base sur des identifiants uniques et immuables. CRATE utilise la fonction d'allocation LSEQ (cf. chapitre 2) afin de fournir ces identifiants. Ainsi, les suppressions suppriment réellement les éléments de la structure d'arbre sous-jacente. De plus, la croissance de ces identifiants est bornée entre une borne logarithmique et une borne polylogarithmique par rapport au nombre d'insertions dans la séquence. Étant sous-linéaire, la complexité en communication devient acceptable. Étant sous-linéaire, la structure d'arbre représentant le document répliqué n'a pas besoin d'être rééquilibrée et donc, l'éditeur n'a pas besoin d'exécuter un protocole de consensus qui ne passerait pas à l'échelle [72].

Le facteur logarithmique de la dissémination de messages, le coût constant apporté par la détection de la causalité et le coût des identifiants implique une complexité en communication attendue entre $\mathcal{O}((\log I) \cdot (\ln R))$ et $\mathcal{O}((\log I)^2 \cdot (\ln R))$ selon les comportements d'édition à l'œuvre, où I est le nombre d'insertions effectuées dans la séquence et R le nombre de répliques dans la session d'édition au moment de la propagation des messages.

4.1.5 Interface utilisateur

Chaque participant possède une réplique du document partagé. Pourtant, l'application doit donner l'illusion d'un unique document accédé par plusieurs utilisateurs.

CRATE est un éditeur temps réel – écrit en HTML, CSS, et JavaScript – qui tourne directement dans les navigateurs Web. Il établit des canaux de communication d'un navigateur à l'autre en utilisant la récente technologie WebRTC [45]. La zone de texte où l'utilisateur peut écrire est fournie grâce à l'éditeur JavaScript Ace [20].

La capture d'écran en figure 4.4 montre l'interface visible par l'utilisateur. Dans cet exemple, au moins trois participants sont impliqués dans la session d'édition. En effet, 3 curseurs sont affichés. Le premier curseur appartient à *Anonymous Mole* et semble être à l'origine du texte *Hello*. Le second curseur appartient à *Anonymous Penguin* et semble être à l'origine du texte *World*. Le troisième curseur est celui de l'utilisateur ayant pris la capture d'écran.

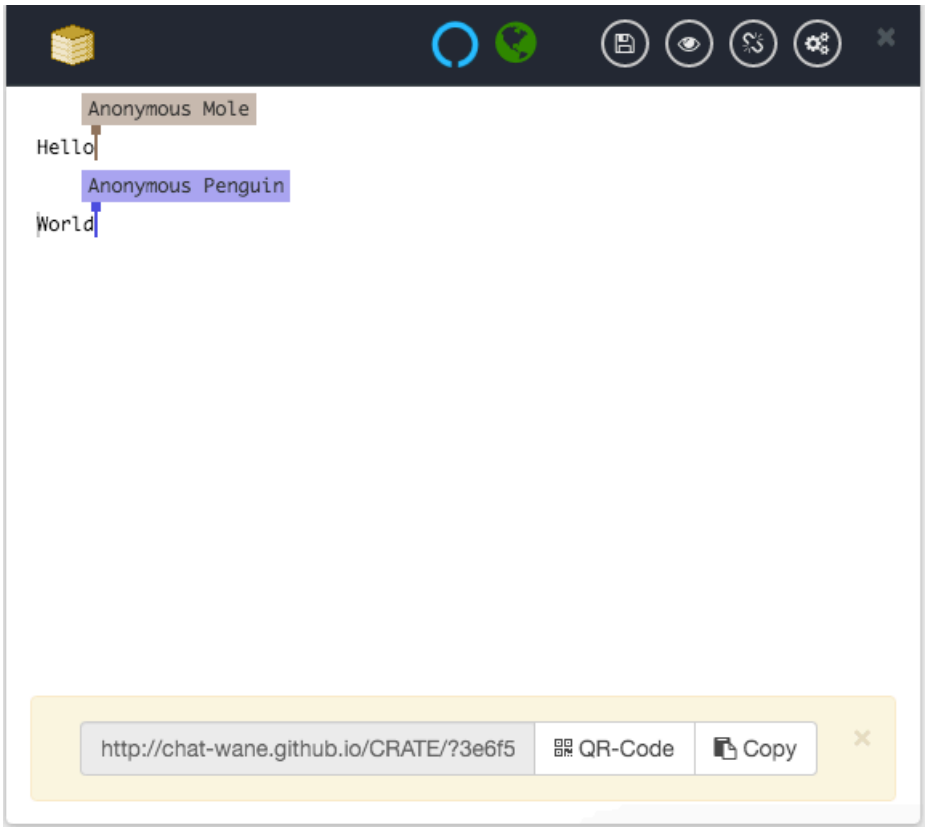


FIGURE 4.4 – Capture d'écran de CRATE.

La barre d'état nous indique que (i) l'éditeur est en train de partager l'accès à la session d'édition via le cercle bleu en rotation. L'utilisateur peut alors donner l'URL en bas de page à d'autres collaborateurs afin qu'ils le rejoignent dans l'écriture du document, en un simple clic ; (ii) que l'utilisateur est bien connecté à d'autres collaborateurs via la planète verte.

Le document lui-même peut contenir des URL référençant d'autres documents édités en temps réel. D'un simple clic, un participant peut naviguer d'une session d'édition temps réel à l'autre de manière décentralisée. Cette simple fonctionnalité permet d'envisager la construction d'un Wikipédia à la fois décentralisé et temps réel.

4.2 Expérimentation

Objectif : Montrer que l'éditeur collaboratif décentralisé Crate passe à l'échelle en termes de nombre d'utilisateurs et de taille de documents.

Description : Cette expérimentation construit des réseaux de 101 à 601 participants. Pour cela, des machines sont réservées sur le banc d'essai Grid'5000. Sur chacune d'elles sont lancés un à plusieurs navigateurs Web observant la procédure d'intégration au réseau. Par exemple, 100 machines avec 6 navigateurs chacune permettent de créer le réseau à 601 participants – le participant supplémentaire étant le créateur du document hébergeant aussi le serveur de signalisation. Chaque session d'édition est en charge d'écrire un document artificiel de plusieurs millions de caractères en insérant ceux-ci à la fin du document. Les mesures concernent le trafic sortant moyen de chacun des membres. Globalement, 100 opérations sont effectuées par seconde, uniformément réparties entre les participants. La session dure 7 heures.

Résultat : La figure 4.5 montre le résultat de cette expérimentation. (i) Le trafic généré croît de façon polylogarithmique par rapport à la taille du document : plus l'expérience progresse, plus le nombre d'insertions effectuées dans le document est important, plus la croissance des identifiants diminue. (ii) À cela s'ajoute un facteur multiplicatif logarithmique par rapport au nombre de participants : plus la session d'édition comporte de membres, plus le trafic généré est important. Toutefois, l'écart entre les mesures diminue tandis que le nombre de participants est incrémenté linéairement. On note toutefois des exceptions à ces observations. Par exemple, les mesures sur 501 éditeurs dominent les mesures sur 601 éditeurs.

Explication : Crate utilise LSeq (cf. chapitre 2). Le comportement d'édition des participants est monotone. LSeq possède une complexité polylogarithmique sur ses identifiants par rapport au nombre d'insertions effectuées dans la séquence. Ainsi, plus les expérimentations progressent, plus le nombre d'insertions dans la séquence augmente. Les messages envoyés ne comportant que les identifiants de LSeq, la croissance du trafic hérite de cette croissance polylogarithmique. Crate utilise Spray (cf. §3). Lorsqu'un nouveau membre

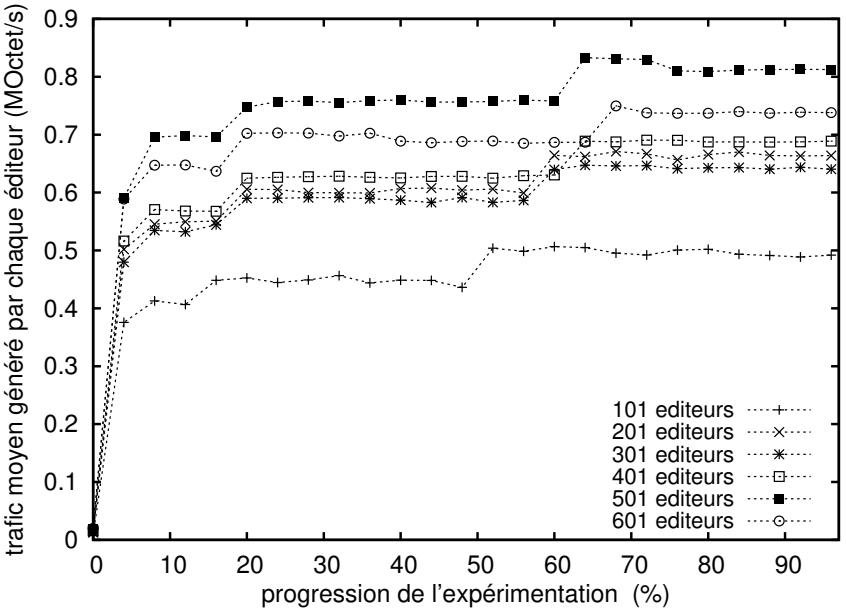


FIGURE 4.5 – Trafic moyen par seconde généré par chaque éditeur pendant la session d'édition. L'axe des abscisses montre la progression de l'expérimentation. L'axe des ordonnées montre le trafic moyen sortant par éditeur en mégaoctets par seconde.

rejoint une session d'édition, il ajoute avec lui un nombre logarithmique de connexions par rapport à la taille du réseau. Chacune de ces connexions est activement utilisée par le mécanisme de diffusion de messages afin que chacun des identifiants parvienne à tous les éditeurs. Pour chaque opération locale effectuée, chaque membre retransmet le message à ses voisins exactement une fois. Ainsi, chaque opération est transmise par chaque nœud à leur vue dont la taille est logarithmique par rapport à la taille de la session d'édition. Ainsi, une session d'édition à 100 participants génère moins de trafic qu'une session d'édition à 200 participants. L'exception perçue entre certaines configurations de l'expérimentation est due à l'aspect aléatoire de la procédure d'entrée dans le réseau de Spray. En effet, le nombre d'arcs ajoutés dans le réseau dépend du nœud contact du nouvel arrivant. Si ce contact possède plus de voisins que le logarithme recherché, alors la taille moyenne des vues partielles augmentera d'autant plus ; et inversement. Sur un grand nombre d'exécutions, l'effet aléatoire induit par les choix de contacts serait gommé.

4.3 Conclusion

Ce chapitre a présenté Crate, un éditeur collaboratif décentralisé fonctionnant directement dans les navigateurs Web. Un utilisateur peut créer, modifier en temps réel et partager son document. À l'instar des éditeurs centralisés, tel que Google Docs, le partage s'effectue très facilement grâce à un simple lien. Lorsqu'un utilisateur clique sur celui-ci, il rejoint la session d'édition et peut à son tour voir, modifier en temps réel et partager le document.

Ce que Crate ne gère pas : la persistance des sessions d'édition. De nos jours, avec l'hégémonie des services hébergés sur le Nuage, il devient plus difficile de supprimer des données que de les sauvegarder. À l'opposé, un document dans Crate n'appartient qu'à ses rédacteurs. La session d'édition et donc le document cessent d'exister lorsque tous les collaborateurs ferment leur éditeur. Si ceux-ci souhaitent sauvegarder le document, ils peuvent le faire localement. Si ceux-ci souhaitent préserver la session d'édition temps réel, ils doivent s'assurer qu'au moins un éditeur reste actif et accessible.

Crate démontre que l'édition collaborative temps réel est possible dans les navigateurs Web, sans l'intervention de tiers et sans limites quant aux dimensions du système.

Le chapitre 5 revient sur les précédentes contributions et présente les perspectives scientifiques ouvertes par celles-ci.

5

Conclusion

Sommaire

5.1 Résumé des contributions

Dans ce manuscrit, notre objectif a été de répondre à la question de faisabilité de l'édition collaborative temps réel dans les navigateurs Web, sans l'intervention de tiers et sans limites quant aux dimensions du système. Nos contributions s'étalent sur les trois points suivants :

Structure de données répliquée. La réplication optimiste permet d'améliorer l'accessiblité, la réactivité et la tolérance aux pannes de la donnée partagée. Pour la séquence, l'état de l'art fait tout d'abord mention des approches à transformées opérationnelles. Cependant, ces approches se voient restreintes aux contextes avec peu de répliques et peu de latence. Plus récemment, les structures dont les opérations commutent ont fait leur apparition. Hélas, soit l'espace nécéssité croît continuellement, même en cas de suppressions d'éléments ; soit celles-ci utilisent des identifiants dont la taille croît linéairement. Ces approches requièrent toutes un mécanisme additionnel afin de recouvrer de bonnes performances. Malheureusement, ces mécanismes sont inutilisables à large échelle. Nous proposons LSeq, une stratégie d'allocation d'identifiants dont les identifiants croissent de manière polylogarith-

mique par rapport au nombre d'insertions effectuées dans la séquence. De plus, les éléments supprimés sont réellement détruit. Ainsi, cette approche reste performante à large échelle et ne nécessite aucun mécanisme additionnel. Nous fournissons l'analyse en complexité – temporelle et spatiale – et en validons les résultats grâce à des expérimentations. LSeq fait l'objet d'une implémentation JavaScript qui est utilisée dans un éditeur collaboratif temps réel décentralisé accessible dans les navigateurs Web.

Communication. Les protocoles d'échantillonnage de pairs constituent le cœur de nombreux systèmes répartis. Ces protocoles proposent à chaque pair une vue partielle du réseau. Les approches fournissant des échantillons aléatoires de pairs convergent vers des topologies proches des graphes aléatoires. Elles garantissent certaines propriétés désirables dans notre contexte telle que la robustesse face à la dynamicité du réseau. L'état de l'art se divise en deux catégories caractérisées par les vues partielles qu'elles fournissent : (i) Les approches dont les vues partielles sont de taille constante définie lors de la configuration sont les plus nombreuses. Malheureusement, le développeur doit prévoir les dimensions des réseaux gérées par son application. Cela n'est pas toujours possible et conduit à un surdimmensionnement des vues partielles. (ii) Les approches dont les vues partielles sont de taille variable ne supportent pas bien les contraintes imposées par la procédure complexe d'établissement de connexion dans les navigateurs Web. Nous proposons une approche, nommée Spray, débarrassant le développeur de ces considérations. La taille des vues partielles suit la taille du réseau. Grâce à elle, les vues partielles sont toujours de taille logarithmique par rapport à la taille du réseau actuel. L'établissement de connexion d'un voisin à l'autre rend l'approche fiable même dans un réseau dynamique. En dépit de ces connexions de proche en proche, la convergence vers un état aux propriétés stables est extrêmement rapide. Ces propriétés ont été validées au travers de simulations. Spray a ensuite été implémenté en JavaScript afin d'être utilisé dans un éditeur collaboratif temps réel décentralisé accessible dans les navigateurs Web.

Éditeur décentralisé dans le navigateur. Google Docs et Etherpad ont rendu l'accès à l'édition collaborative aisé pour des millions d'utilisateurs. Toutefois, la centralisation de ces services pose des problèmes de confidentialité, de passage à l'échelle et de tolérance aux défaillances. Nous proposons Crate, un éditeur collaboratif temps réel décentralisé tournant directement dans les navigateurs Web. Étant décentralisé, un document appartient seulement à ceux qui l'éditent, ce qui règle les problèmes de confidentialité. Utilisant Spray et LSeq, Crate parvient à adapter le trafic généré à la session d'édition. Ce trafic croît logarithmiquement par rapport à la taille de la session d'édition. Ce trafic croît polylogarithmiquement par rapport au nombre d'insertions dans le document. Ce résultat est validé par des expérimentations impliquant jusqu'à 601 navigateurs écrivant un document artificiel de plusieurs millions de caractères.

5.2 Perspectives

Tout d'abord, chacun des composants de notre éditeur collaboratif peut être amélioré ou étendu. Ensuite, l'éditeur lui-même offre des opportunités inédites. Par exemple, son intégration, non en opposition, mais en complément des approches actuellement centralisés. Enfin, l'architecture ouvre aussi la voie à un plus large champ d'applications décentralisées directement accessibles via les navigateurs Web. Cette section fournit une liste de perspectives scientifiques concernant ces travaux de thèse.

5.2.1 Fusion de réseaux

La fusion de réseaux consiste à obtenir un réseau unique comme l'union des membres de plusieurs réseaux. Le réseau obtenu doit hériter des propriétés de ses parents. Lors de cette procédure, nous supposons qu'au moins un des nœuds appartenant à l'un des réseau contacte l'autre réseau afin d'initier la fusion. Grâce à la connexion qui en résulte, les réseaux sont à même de communiquer et donc de fusionner.

Les approches à taille fixe sont triviales à étendre : les mélanges périodiques suffisent à garantir un réseau connexe. Si toutefois les vues partielles sont configurées avec des tailles différentes, il suffit alors de prendre la taille maximum des deux. Par exemple, un nœud avec une vue partielle de 5 voisins verra sa vue augmenter à 7 voisins après un échange avec le nœud dont la vue partielle est peuplée de 7 références.

Spray est une approche dont les vues partielles évoluent automatiquement en réaction aux entrées et sorties du réseau. En particulier, les vues partielles suivent une progression logarithmiques par rapport à la taille du réseau. La fusion de réseaux Spray doit résulter en un réseau Spray garantissant cette progression.

La première solution venant à l'esprit est la suivante : chacun des nœuds du premier réseau utilise le contact afin de rejoindre le second réseau, comme un sablier dont les grains passent un tube étroit pour rejoindre l'autre bulbe sous l'effet de la gravité. Malheureusement, cette solution est extrêmement lente – puisque la majorité des nœuds ignorent encore qui est le contact – et susceptible d'échouer – puisque le contact est un point unique de défaillance.

Un seconde solution consiste simplement, à l'instar des approches à taille fixe, à laisser le mécanisme de mélange faire son office. Progressivement, d'autres ponts entre les réseaux vont se former jusqu'à ce que les deux réseaux soient indifférenciés. Malheureusement, les arcs ne suivent pas l'augmentation relative au réseau.

Définition du problème 3. *Soit $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_k$ des réseaux de taille arbitraire. On a :*

$$\sum_{i=1}^{k} |\mathcal{N}_i| \ln(|\mathcal{N}_i|) < (\sum_{i=1}^{k} |\mathcal{N}_i|) \ln (\sum_{i=1}^{k} |\mathcal{N}_i|) \tag{5.1}$$

Comment adapter le nombre d'arcs effectifs (à gauche) pour qu'il atteigne le nombre d'arcs requis (à droite) ?

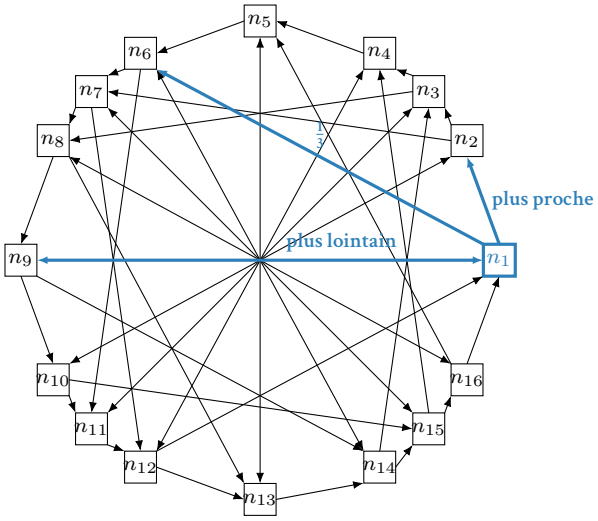


FIGURE 5.1 – Table de hachage répartie suivant le principe de Chord.

5.2.2 Table de hachage répartie

Une table de hachage répartie (*DHT*[1]) est un système permettant de retrouver une ressource rapidement dans un réseau superposé de nœuds. La ressource recherchée possède une clé permettant l'exploration efficace du réseau.

Parmi les représentants des DHT on trouve Chord [93], CAN [85], Pastry [87], Tapestry [113], Kademlia [64] ou Kelips [44]. Ces approches font souvent état d'une vue partielle logarithmique. Toutefois, tout comme pour les protocoles d'échantillonnage aléatoire de pairs dont la vue partielle est fixe, tel que CYCLON, la taille de leur vue est configurée par avance. Lorsque le système fait face à de fréquentes entrées et sorties de nœuds, les vues peuvent être ajustées grâce à des mécanismes d'estimation de la taille du réseau [17, 38]. Cependant, ces mécanismes présentent un coût additionnel [38].

Construire une DHT au dessus d'un protocole d'échantillonnage aléatoire de pairs apporte le double avantage d'une convergence rapide vers une topologie optimale et d'une résilience aux fréquentes entrées et sorties de nœuds [54, 69, 100]. Comme le montre la figure 5.1, l'idée est de placer les systèmes classiques de DHT – ici Chord – afin qu'ils s'accordent avec la vue partielle fournie par SPRAY. Par exemple, si un nœud SPRAY possède 3 voisins, le réseau superposé en charge de la DHT possède lui aussi 3 voisins. Toutefois, ces derniers sont choisis selon la distance à laquelle ils se trouvent. Un premier voisin serait celui qui est le plus proche, le second voisin celui qui se trouve à la moitié de la distance maximum et le troisième se trouverait à un tiers de la distance maximum. Un tel système

[1] *Distributed Hash Table.*

permet de diriger les messages ciblant un nœud particulier très efficacement : de l'ordre de $\log |\mathcal{N}|$ sauts en moyenne, où $\mathcal{N}$ est l'ensemble des membres appartenant au réseau.

5.2.3 Compromis causalité et concurrence

CRATE comprend une couche dédiée à la détection de relations causales. La structure utilisée est celle d'un vecteur d'horloges incluant des exceptions [62]. Celle-ci, identiquement aux vecteurs d'horloges, stockent localement au moins un entier par collaborateur ayant jamais participé dans l'édition. Pour les dispositifs informatiques aux configurations plus modestes comme les téléphones portables, la progression linéaire de ces vecteurs constitue un problème.

Une perspective possible serait de remplacer ce vecteur d'horloges de taille W, où W est le nombre de membres ayant jamais participé à la session d'édition, par un vecteur d'horloges de taille K, où K est un entier nettement inférieur à W. L'intuition derrière cette structure provient du fait que lorsqu'il n'y a pas de concurrence, une seule horloge de Lamport [56] suffit pour caractériser les relations causales. Si la concurrence augmente, alors des doublons peuvent apparaître. L'intégration, en présence de doublons, peut conduire à des incohérences.

L'idée serait alors d'allier (i) une structure utilisant un vecteur dont la taille s'ajuste à la concurrence du système afin de borner la fréquence des erreurs ; (ii) un mécanisme de recouvrement sur erreur afin d'intégrer l'opération arrivée en retard ; (iii) un mécanisme d'anti-entropie afin que le système soit fiable : toutes les opérations parviennent à tous les participants.

5.2.4 O'Browser, Where Art Thou ?

De nos jours, les navigateurs Web sont plus que de simples visualisateurs de contenu, ils sont presque devenus des systèmes d'exploitation. Ils comptent parmi les programmes les plus distribués de par le monde. Ils apparaissent dans un large éventail d'appareils aux diverses capacités tels que les mobiles, les tablettes ou les ordinateurs de bureaux.

Développer une application Web, c'est développer pour une large audience hétérogène. L'éditeur collaboratif temps réel CRATE prouve qu'il est possible de développer des applications décentralisées directement dans le navigateur Web. Le projet WebTorrent [2] constitue un autre exemple d'application décentralisée fonctionnant dans le navigateur Web. Ce projet permet le transfert de fichiers statique. Quelles autres applications décentralisées est-il possible de développer ?

Rassembler l'ensemble des composants répartis en un langage adapté au Web permettrait de développer un catalogue d'applications décentralisées garantissant certaines propriétés sur, par exemple, le critère de cohérence ou la sécurité. CRATE ne serait que l'une des applications temps réel que ce langage permettrait d'écrire. Cela constituerait une avancée

certaine en faveur d'un Web décentralisé. De surcroît, lorsque des navigateurs Web sont actuellement entièrement développés dans l'optique de supporter le décentralisé [14] – ce qui représente une dépense substantielle – le même objectif pourrait être atteint en étendant simplement les capacités des navigateurs Web existants.

Bibliographie

[1] Saleh E. Abdullahi and Graem A. Ringwood. Garbage collecting the internet : A survey of distributed garbage collection. *ACM Comput. Surv.*, 30(3) :330–373, September 1998.

[2] Feross Aboukhadijeh. Webtorrent. `https://webtorrent.io`.

[3] Mehdi Ahmed-Nacer. *Evaluation methodology for replicated data types.* Theses, Université de Lorraine, May 2015.

[4] Mehdi Ahmed-Nacer, Claudia-Lavinia Ignat, Gérald Oster, Hyun-Gul Roh, and Pascal Urso. Evaluating CRDTs for Real-time Document Editing. In ACM, editor, *11th ACM Symposium on Document Engineering*, pages 103–112, Mountain View, California, États-Unis, September 2011.

[5] Nathan Jenkins Alberto Pepe and Matteo Cantiello. Authorea. `https://www.authorea.com`.

[6] Amazon. Amazon dynamodb. `http://aws.amazon.com/fr/dynamodb/`.

[7] Luc André, Stéphane Martin, Gérald Oster, and Claudia-Lavinia Ignat. Supporting Adaptable Granularity of Changes for Massive-scale Collaborative Editing. In *CollaborateCom - 9th IEEE International Conference on Collaborative Computing : Networking, Applications and Worksharing - 2013*, Austin, États-Unis, October 2013.

[8] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. Specification and complexity of collaborative text editing. In *Proceedings of the 2016 ACM symposium on Principles of distributed computing*. ACM – Association for Computing Machinery, July 2016.

[9] Peter Bailis and Ali Ghodsi. Eventual consistency today : Limitations, extensions, and beyond. *Commun. ACM*, 56(5) :55–63, May 2013.

[10] Basho. Riak. `http://fr.basho.com/products/`.

[11] Juan Benet. IPFS - content addressed, versioned, P2P file system. *CoRR*, abs/1407.3561, 2014.

[12] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. *Brief Announcement : Semantics of Eventually Consistent Replicated Sets*, pages 441–442. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[13] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2) :41–88, May 1999.

[14] inc BitTorrent. Project maelstrom. `http://project-maelstrom.bittorrent.com`.

[15] Lori B. Breslow, David E. Pritchard, Jennifer DeBoer, Glenda S. Stump, Andrew D. Ho, and Daniel T. Seaton. Studying learning in the worldwide classroom : Research into edx's first mooc. *Research & Practice in Assessment*, 8 :13–25, 2013.

[16] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types : Specification, verification, optimality. *SIGPLAN Not.*, 49(1) :271–284, January 2014.

[17] Gonzalo Camarillo and Jouni Maenpaa. Self-tuning distributed hash table (dht) for resource location and discovery (reload). RFC 7363, Ericsson, September 2014.

[18] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1) :11–16, July 1991.

[19] Ronan-Alexandre Cherrueau. *Composer les techniques de sécurité pour un nuage plus sûr*. PhD thesis, École des Mines de Nantes, 2016.

[20] Cloud9 IDE and Mozilla. Ace. `https://ace.c9.io`.

[21] Neil Conway. *Language Support for Loosely Consistent Distributed Programming*. PhD thesis, University of California, Berkeley, 2014.

[22] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi : A decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.*, 34(4) :15–26, August 2004.

[23] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.

[24] Gerardine Desanctis and R. Brent Gallupe. A foundation for the study of group decision support systems. *Management Science*, 33(5) :589–609, May 1987.

[25] Clarence A. Ellis and Simon J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, SIGMOD '89, pages 399–407, New York, NY, USA, 1989. ACM.

[26] Clarence A. Ellis, Simon J. Gibbs, and Gail Rein. Groupware : some issues and experiences. *Communications of the ACM*, 34(1) :39–58, 1991.

[27] Douglas C. Engelbart and William K. English. A research center for augmenting human intellect. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 395–410, New York, NY, USA, 1968. ACM.

[28] Paul Erdős and Alfréd Rényi. On random graphs i. *Publ. Math. Debrecen*, 6 :290–297, 1959.

[29] Patrick Th. Eugster, Rachid Guerraoui, Sidath B. Handurukande, Petr Kouznetsov, and Anne-Marie Kermarrec. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems (TOCS)*, 21(4) :341–374, November 2003.

[30] Pauline Folz, Hala Skaf-Molli, and Pascal Molli. *The Semantic Web. Latest Advances and New Domains : 13th International Conference, ESWC 2016, Heraklion, Crete, Greece, May 29 – June 2, 2016, Proceedings*, chapter CyCLaDEs : A Decentralized Cache for Triple Pattern Fragments, pages 455–469. Springer International Publishing, Cham, 2016.

[31] Apache Software Foundation. Cassandra. `http://cassandra.apache.org`.

[32] Etherpad Foundation. Etherpad. `http://etherpad.org`.

[33] Linux Foundation. netem. `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`.

[34] Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, Boris Koldehofe, Martin Mogensen, Maxime Monod, and Vivien Quéma. Heterogeneous gossip. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 3 :1–3 :20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[35] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Scamp : Peer-to-peer lightweight membership service for large-scale group communication. In Jon Crowcroft and Markus Hofmann, editors, *Networked Group Communication*, volume 2233 of *Lecture Notes in Computer Science*, pages 44–55. Springer Berlin Heidelberg, 2001.

[36] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2) :139–149, Feb 2003.

[37] Barton Gellman and Laura Poitras. U.S., british intelligence mining data from nine U.S. internet companies in broad secret program, June 2013. washingtonpost.com [Online ; posted 7-June-2013].

[38] Gabriel Ghinita and Yong Meng Teo. An adaptive stabilization framework for distributed hash tables. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pages 10 pp.–, April 2006.

[39] Jim Giles. Internet encyclopaedias go head to head. *Nature*, 438(7070) :900–901, Dec 2005.

[40] Google. Google docs. `https://docs.google.com`.

[41] Google. Google wave. `http://wave.google.com`.

[42] Victor Grishchenko. Deep hypertext with embedded revision control implemented in regular expressions. In *Proceedings of the 6th International Symposium on Wikis and Open Collaboration*, WikiSym '10, pages 3 :1–3 :10, New York, NY, USA, 2010. ACM.

[43] Jonathan Grudin. Computer-supported cooperative work : History and focus. *Computer*, 27(5) :19–26, May 1994.

[44] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse. *Peer-to-Peer Systems II : Second International Workshop, IPTPS 2003, Berkeley, CA, USA, February 21-22, 2003. Revised Papers*, chapter Kelips : Building an Efficient and Stable P2P DHT through Increased Memory and Background Overhead, pages 160–169. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[45] IETF and W3C. Webrtc. `http://www.webrtc.org`.

[46] MongoDB inc. Mongodb. `https://www.mongodb.com`.

[47] Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The peer sampling service : Experimental evaluation of unstructured gossip-based implementations. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '04, pages 79–98, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[48] Márk Jelasity and Alberto Montresor. Epidemic-style proactive aggregation in large overlay networks. In *Proceedings. 24th International Conference on Distributed Computing Systems, 2004*, pages 102–109, 2004.

[49] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-man : Gossip-based fast overlay topology construction. *Computer Networks*, 53(13) :2321 – 2339, 2009. Gossiping in Distributed Systems.

[50] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3) :8, 2007.

[51] Robert Johansen. *GroupWare : Computer Support for Business Teams.* The Free Press, New York, NY, USA, 1988.

[52] Paul R. Johnson and Robert H. Thomas. Maintenance of duplicate databases. RFC 677, January 1975.

[53] Marcel Klehr. Hive.js. `http://hivejs.org`.

[54] Sveta Krasikova, Raziel C. Gómez, Heverson B. Ribeiro, Etienne Rivière, and Valerio Schiavoni. *Distributed Applications and Interoperable Systems : 16th IFIP WG 6.1 International Conference, DAIS 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, chapter Evaluating the Cost and Robustness of Self-organizing Distributed Hash Tables, pages 16–31. Springer International Publishing, Cham, 2016.

[55] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4) :360–391, November 1992.

[56] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7) :558–565, July 1978.

[57] Janne Lautamäki. *On the Development of Real-Time Multi-User Web Applications.* PhD thesis, Tampere University of Technology, 2013.

[58] Janne Lautamäki, Antti Nieminen, Johannes Koskinen, Timo Aho, Tommi Mikkonen, and Marc Englund. Cored : Browser-based collaborative real-time editor for java web applications. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1307–1316, New York, NY, USA, 2012. ACM.

[59] Joao Leitão, José Pereira, and Luis Rodrigues. Hyparview : A membership protocol for reliable gossip-based broadcast. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pages 419–429, June 2007.

[60] Mihai Letia, Nuno Preguiça, and Marc Shapiro. Crdts : Consistency without concurrency control. *Arxiv preprint arXiv :0907.0929*, 2009.

[61] Writelatex Limited. Overleaf. `https://www.overleaf.com`.

[62] Dahlia Malkhi and Doug Terry. Concise version vectors in winfs. *Distributed Computing*, 20(3) :209–219, 2007.

[63] Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Ashraf Aboulnaga, and Tim Berners-Lee. A demonstration of the solid platform for social web applications. In *Proceedings of the 25th International Conference Companion on World Wide Web*, WWW '16 Companion, pages 223–226, Republic and Canton of Geneva, Switzerland, 2016. International World Wide Web Conferences Steering Committee.

[64] Petar Maymounkov and David Mazières. Kademlia : A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[65] Ahmed-Nacer Mehdi, Pascal Urso, Valter Balegas, and Nuno Perguiça. Merging ot and crdt algorithms. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, pages 9 :1–9 :4, New York, NY, USA, 2014. ACM.

[66] Peter Mell and Timothy Grance. Sp 800-145. the nist definition of cloud computing. Technical report, National Institute of Standards and Technology, Gaithersburg, MD, United States, 2011.

[67] Microsoft. Microsoft word. `https://products.office.com/fr-fr/word`.

[68] Stephen Mogan and Weigang Wang. The impact of web 2.0 developments on real-time groupware. In *2010 IEEE Second International Conference on Social Computing (SocialCom)*, pages 534–539, Aug 2010.

[69] A. Montresor, M. Jelasity, and O. Babaoglu. Chord on demand. In *Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*, pages 87–94, Aug 2005.

[70] Alberto Montresor and Márk Jelasity. Peersim : A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, September 2009.

[71] Alberto Montresor, Márk Jelasity, and Ozalp Babaoglu. Robust aggregation protocols for large-scale overlay networks. In *2004 International Conference on Dependable Systems and Networks*, pages 19–28, June 2004.

[72] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-Free Asynchronous Binary Byzantine Consensus with $t< n/3$, $O(n^2)$ Messages, and $O(1)$ Expected Time. *Journal of the ACM (JACM)*, 62 :1000–1020, December 2015.

[73] Madhavan Mukund, Gautham Shenoy R., and S.P. Suresh. Optimized or-sets without ordering constraints. In Mainak Chatterjee, Jian-nong Cao, Kishore Kothapalli, and Sergio Rajsbaum, editors, *Distributed Computing and Networking*, volume 8314 of *Lecture Notes in Computer Science*, pages 227–241. Springer Berlin Heidelberg, 2014.

[74] Brice Nédelec, Pascal Molli, and Achour Mostefaoui. Crate : Writing stories together with our browsers. In *Proceedings of the 25th International Conference Companion on World Wide Web*, WWW '16 Companion, pages 231–234, Republic and Canton of Geneva, Switzerland, 2016. International World Wide Web Conferences Steering Committee.

[75] Brice Nédelec, Pascal Molli, Achour Mostéfaoui, and Emmanuel Desmontils. Concurrency effects over variable-size identifiers in distributed collaborative editing. In *DChanges*, volume 1008 of *CEUR Workshop Proceedings*. CEUR-WS.org, September 2013.

[76] Brice Nédelec, Pascal Molli, Achour Mostéfaoui, and Emmanuel Desmontils. LSEQ : an Adaptive Structure for Sequences in Distributed Collaborative Editing. In ACM, editor, *13th ACM Symposium on Document Engineering*, September 2013.

[77] Brice Nédelec, Julian Tanke, Davide Frey, Pascal Molli, and Achour Mostefaoui. Spray : an Adaptive Random Peer Sampling Protocol. Technical report, LINA-University of Nantes ; INRIA Rennes - Bretagne Atlantique, September 2015.

[78] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. *Engineering the Web in the Big Data Era : 15th International Conference, ICWE 2015, Rotterdam, The Netherlands, June 23-26, 2015, Proceedings*, chapter Yjs : A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types, pages 675–678. Springer International Publishing, Cham, 2015.

[79] Sylvie Noël and Jean-Marc Robert. Empirical study on collaborative writing : What do co-authors do, use, and like ? *Computer Supported Cooperative Work (CSCW)*, 13(1) :63–89, 2004.

[80] Gerald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for p2p collaborative editing. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 259–268. ACM, 2006.

[81] Henry Oswald and James Allen. Sharelatex. `https://fr.sharelatex.com`.

[82] Siani Pearson. Toward accountability in the cloud. *IEEE Internet Computing*, 15(4) :64–69, July 2011.

[83] Jeffrey M Perkel. Scientific writing : the online cooperative. *Nature*, 514(7520) :127–128, 2014.

[84] Nuno Preguiça, Joan M. Marquès, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 395–403. Ieee, 2009.

[85] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 161–172, New York, NY, USA, 2001. ACM.

[86] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types : Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3) :354–368, 2011.

[87] Antony I. T. Rowstron and Peter Druschel. Pastry : Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, 2001. Springer-Verlag.

[88] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1) :42–81, March 2005.

[89] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, January 2011.

[90] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, 2011.

[91] Fog Creek Software. Hyperdev. `https://hyperdev.com/`.

[92] Richard Stallman and Guy Steele. Emacs. `https://www.gnu.org/software/emacs/`.

[93] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[94] Chengzheng Sun and 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*, CSCW '98, pages 59–68, New York, NY, USA, 1998. ACM.

[95] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1) :63–108, 1998.

[96] David Sun and Chengzheng Sun. Context-based operational transformation in distributed collaborative editing systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(10) :1454–1470, Oct 2009.

[97] Norbert Tölgyesi and Márk Jelasity. Adaptive peer sampling with newscast. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 523–534. Springer Berlin Heidelberg, 2009.

[98] Albert van der Linde, João Leitão, and Nuno Preguiça. Δ-crdts : Making δ-crdts delta-based. In *Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data*, PaPoC '16, pages 12 :1–12 :4, New York, NY, USA, 2016. ACM.

[99] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. Cyclon : Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2) :197–217, 2005.

[100] Spyros Voulgaris and Maarten van Steen. *Middleware 2013 : ACM/IFIP/USENIX 14th International Middleware Conference, Beijing, China, December 9-13, 2013, Proceedings*, chapter VICINITY : A Pinch of Randomness Brings out the Structure, pages 21–40. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[101] Jimmy Wales and Larry Sanger. Wikipedia. `https://en.wikipedia.org`.

[102] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *nature*, 393(6684) :440–442, 1998.

[103] Stéphane Weiss. *Collaborative editing over P2P networks.* Theses, Université Henri Poincaré - Nancy I, October 2010.

[104] Stéphane Weiss, Pascal Urso, and Pascal Molli. Wooki : A p2p wiki-based collaborative writing tool. In Boualem Benatallah, Fabio Casati, Dimitrios Georgakopoulos,

Claudio Bartolini, Wasim Sadiq, and Claude Godart, editors, *Web Information Systems Engineering – WISE 2007*, volume 4831 of *Lecture Notes in Computer Science*, pages 503–512. Springer Berlin Heidelberg, 2007.

[105] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot : a scalable optimistic replication algorithm for collaborative editing on p2p networks. In *ICDCS'09. 29th IEEE International Conference on Distributed Computing Systems, 2009*, pages 404–412. IEEE, 2009.

[106] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot-undo : Distributed collaborative editing system on p2p networks. *IEEE Transactions on Parallel Distributed Systems*, 21(8) :1162–1174, 2010.

[107] Gavin Wood. Ethereum : A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.

[108] Fidus Writer. Fidus writer. `https://www.fiduswriter.org`.

[109] Qinyi Wu, Calton Pu, and João Eduardo Ferreira. A partial persistent data structure to support consistency in real-time collaborative editing. In *IEEE 26th International Conference on Data Engineering (ICDE), 2010*, pages 776–779, 2010.

[110] Weihai Yu. A string-wise crdt for group editing. In *Proceedings of the 17th ACM international conference on Supporting group work*, GROUP '12, pages 141–144, New York, NY, USA, 2012. ACM.

[111] Marek Zawirski. *Dependable Eventual Consistency with Replicated Data Types*. PhD thesis, L'université Pierre et Marie Curie (UPMC), Paris, France, January 2015.

[112] Marek Zawirski, Marc Shapiro, and Nuno Preguiça. Asynchronous rebalancing of a replicated tree. In *Conf. Française de Systèmes d'Exploitation (CFSE)*, page 12, Saint-Malo, France, May 2011.

[113] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry : A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1) :41–53, September 2006.

UNIVERSITE BRETAGNE LOIRE

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Brice NÉDELEC

Édition collaborative décentralisée dans les navigateurs

Decentralized Collaborative Editing in Web Browsers

Résumé
Un éditeur collaboratif permet de répartir la tâche de rédaction d'un document à travers le temps et l'espace. Par leur simplicité d'utilisation, les éditeurs collaboratifs temps réel du Web ont contribué à l'adoption massive de ces outils par le grand public. Cependant, les éditeurs actuels sont centralisés : un serveur appartenant à un fournisseur de services gère une session d'édition. En résultent des problèmes de confidentialité, de censure, de propriété, de passage à l'échelle et de tolérance aux pannes.
Récemment, la possibilité d'établir des communications d'un navigateur Web à l'autre a ouvert de nouvelles opportunités en faveur d'un Web décentralisé. Un éditeur collaboratif temps réel décentralisé fonctionnant dans les navigateurs Web doit gérer efficacement des groupes de taille variable et hautement dynamiques.
Cette thèse comporte trois contributions : (i) Pour représenter le document, nous proposons une structure de données répliquée dont la taille des métadonnées croît de manière sous-linéaire par rapport au nombre de caractères insérés dans le document. (ii) Pour propager efficacement les changements à tous les éditeurs participant à l'édition, nous proposons un protocole d'échantillonnage aléatoire de pairs adapté aux contraintes des navigateurs Web et s'ajustant automatiquement au logarithme de la taille de la session d'édition. (iii) Pour démontrer la faisabilité d'un éditeur collaboratif temps réel décentralisé fonctionnant dans les navigateurs Web, nous proposons un éditeur réunissant (i) et (ii), et dont les performances passent à l'échelle.

Mots clés
Édition collaborative, décentralisé, temps réel, Web, passage à l'échelle, structure de données répartie pour séquences, échantillonnage aléatoire de pairs.

Abstract
Collaborative editors allow users to distribute the writing of a document across space and time. Thanks to their ease of use, real-time collaborative editors working in Web browsers vastly contributed to the adoption of such tools. However, current editors are centralized : a service provider's server hosts an editing session. It raises privacy and scalability issues.
Recently, the enabling of browser-to-browser connection establishments opened new opportunities in favor of a decentralized Web. Decentralized real-time collaborative editors working in Web browsers must efficiently handle highly dynamic groups of different size.
Contributions of this thesis are threefold : (i) To represent the document, we propose a replicated data structure for sequences using metadata the size of which scales sublinearly compared to the number of inserted characters. (ii) To efficiently propagate the changes to all editors involved in the collaborative writing, we propose a random peer sampling protocol that supports Web browsers constraints and self-adjusts its functioning to the variations of network membership. (iii) To demonstrate the feasibility of a decentralized real-time collaborative editors running in Web browsers, we propose an editor using (i) and (ii), and we highlight its scalability.

Key Words
Collaborative editing, decentralized, real-time, Web, scalable, replicated structure for sequences, random peer sampling.