UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Réplication sécurisée dans les infrastructures pair-à-pair de collaboration

THÈSE

présentée et soutenue publiquement le 14 juin 2021

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Victorien Elvinger

Composition du jury

Président :	Steve Kremer	Directeur de recherche, Inria Grand Est
Rapporteur·ice·s :	Emmanuelle Anceaume	Directrice de recherche, IRISA
	Pascal Molli	Professeur des universités, Université de Nantes
Examinatrice :	Esther Pacitti	Professeure des universités, Université de Montpellier 2
Encadrants :	François Charoy	Professeur des universités, Université de Lorraine
	Gérald oster	Maître de conférence, Université de Lorraine

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

Mis en page avec la classe thesul.

Remerciements

Avant d'être une épreuve intellectuelle, le doctorat est d'abord une épreuve émotionnelle. Cet aspect est souvent sous-estimé ou invisibilisé. Les études sur le sujet existent et mettent en exergue le mal-être qui touche les doctorants et les doctorantes de toutes disciplines. Profondément attaché à l'amélioration de la société, je fais le vœu que le doctorat soit à l'avenir plus sain ou, à défaut, cède sa place à d'autres formes d'aventures intellectuelles et humaines.

Ceci dit, mon doctorat a été une aventure intellectuelle stimulante dans laquelle j'ai pu nourrir ma curiosité. Cette aventure a été l'occasion de nouer des liens avec mes encadrants François Charoy et Gérald Oster. Je les remercie de m'avoir témoigné leur confiance et de m'avoir donné carte blanche pour mener à bien cette thèse. Je remercie également Claudia-Lavinia Ignat, chargée de recherche à Inria, pour les échanges constructifs que nous avons eus.

Que serait ma thèse sans l'amitié et la bienveillance de mes camarades de travail ? Je pense en particulier à Matthieu Nicolas avec qui j'ai eu des échanges critiques et qui a été d'un soutien infaillible. Je pense à Quentin Laporte Chabasse avec qui j'ai eu des débats passionnés. Je pense à Phillippe Kalitine, à Linda Ouchaou, et à Anis Ahmed Nacer qui ont chaque jour apporté leur niaque. Je pense à Hoai-Le Nguyen qui a partagé avec modestie sa passion de la raquette. Je pense également à Abir Ismaili-Alaoui, Alexandre Bourbeillon, Amina Abdmeziem, Béatrice Linot, Cédric Enclos, Chahrazed Labba, Clélie Amiot, Guillaume Rosinosky, Hoang Long Nguyen, Jean-Philippe Eisenbarth, Pierre-Antoine Rault, Riyadh Abdmeziem, et Siavash Atarodi.

De nombreuses personnes ont contribué indirectement à cette thèse et de manière très diverse. Je pense aux employés du restaurant d'Inria Grand Est qui apportent chaque jour leur énergie pour faire sourire et proposer des repas de qualité. Je pense aux employés administratifs qui m'ont permis de mener à bien les procédures qui jonchent le parcours du doctorat. Je pense au personnel d'entretien des bâtiments qui préservent un cadre de travail propre et fonctionnel.

Je souhaiterais remercier mes ami·e·s de longue date Alexandre Merlin, Alizée Dulliand, et Rado Randrianomanana pour avoir été à mes côtés durant ces années de thèse. Je remercie également ma sœur Alice Elvinger, mes parents Clarisse Elvinger et Jean-Michel Elvinger, mes grands-parents maternels Marie-Thérèse Mollet et Michel Mollet, mes grands-parents paternels Jeanine Elvinger et Claude Elvinger, ainsi que mes grands-parents de cœur Bernadette Petitcolas et René Petitcolas. Ils et elles ont tous et toutes contribué à ce que je suis aujourd'hui.

Je dédie cette thèse à toute personne désireuse d'un monde plus libre et plus juste.

Table des matières

Liste des figures

Liste des tableaux

Liste des définitions et théorèmes

Liste des algorithmes

Chapitre 1

Introduction

Sommaire

Ce premier chapitre introduit les infrastructures de collaboration. Il motive notre intérêt pour les infrastructures pair-à-pair de collaboration. Il donne un aperçu de nos problématiques de recherche et de nos contributions. Il détaille également l'organisation de ce manuscrit et il énumère nos publications.

1.1 Contexte

Internet est devenu une plate-forme où les internautes ne consultent plus uniquement des contenus, mais les produisent et les enrichissent collectivement [1] : elles et ils partagent et commentent des contenus multimédias au sein des réseaux sociaux ; elles et ils co-écrivent des documents à l'aide d'éditeurs collaboratifs ; elles et ils co-programment à l'aide de gestionnaires de versions. Les productions de tels contenus sont le résultat d'activités de collaboration [2] qui se répartissent dans l'espace et le temps. Les contributeur·ice·s modifient individuellement un contenu partagé. Elles et ils peuvent se trouver à des endroits géographiquement éloignés et peuvent modifier simultanément ou de manière différée les contenus partagés. Leurs modifications sont à terme observées et prises en compte par les autres contributeur·ice·s.

Les applications de collaboration ont permis l'émergence et la diffusion des activités de collaboration sur Internet. Ces applications ont été initialement conçues sur des infrastructures centralisées. Dans de telles infrastructures, les activités de collaboration s'organisent autour d'un serveur. Les collaborateur·ice·s sont connecté·e·s les un·e·s aux autres à travers le serveur. Toutes les modifications qu'elles et ils effectuent sur les contenus partagés sont relayées par le serveur aux autres collaborateur·ice·s. Le serveur peut éventuellement transformer les modifications avant de les transmettre. Dans la figure 1.1 les collaboratrices soumettent leur modification au serveur qui les appliquent sur le contenu. Le serveur transmet la nouvelle version du contenu aux collaboratrices. Le rôle prépondérant du serveur engendre plusieurs problèmes :

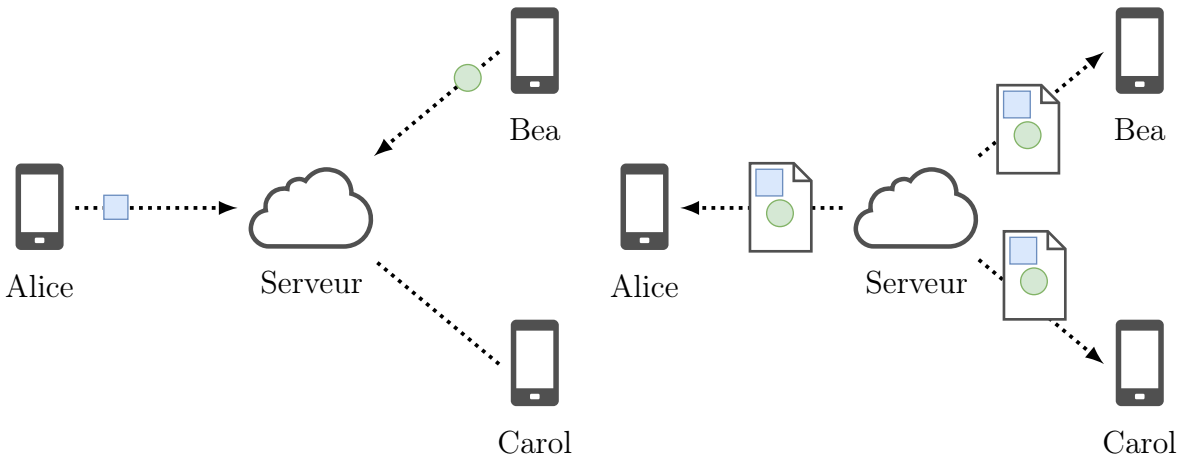


FIGURE 1.1 – Exemple d'une activité de collaboration au sein d'une infrastructure centralisée. Alice, Bea, et Carol sont connectées à un serveur. Elles éditent ensemble un dessin. (a) Alice et Bea souhaitent modifier le dessin partagé : Alice souhaite ajouter un carré et Bea souhaite ajouter un cercle. Elles soumettent leur modification au serveur. (b) Le serveur intègre leurs modifications et transmet la nouvelle version du dessin aux collaboratrices.

Disponibilité. Toutes les modifications du contenu partagé sont transmises aux collaborateur·ice·s par l'intermédiaire du serveur. L'indisponibilité du serveur rend impossible la collaboration. Le serveur est un *point unique de défaillance* [3].

Latence. Le traitement centralisé des modifications effectuées par les collaborateur·ice·s introduit des latences. Ces latences, ajoutées à celles du réseau, peuvent être trop importantes pour la modification de contenu en simultané. Nous parlons de simultané quand des collaborateur·ice·s éditent un même contenu au même moment et qu'elles et ils ont l'illusion de percevoir immédiatement les modifications de chacun·e. IGNAT et al. [4] ont montré que les activités de collaboration en simultané étaient compromises lorsque le délai entre des modifications de contenu et leurs observations par les autres était trop important. Ce délai engendre des conflits de modification sur le contenu partagé. Les collaborateur·ice·s adoptent des stratégies qui limitent leurs interactions pour éviter ces conflits. Ce qui réduit l'efficacité de la collaboration.

Passage à l'échelle. Le serveur peut difficilement s'adapter à un accroissement du nombre de modifications et du nombre de collaborateur·ice·s. Cette difficulté se traduit par une augmentation des latences pour intégrer de nouvelles modifications et propager les nouvelles versions du contenu partagé. Elle peut également produire un déni de service qui rend indisponible le serveur. Pour éviter ce problème les applications de collaboration limitent le nombre de collaborateur·ice·s autorisé·e·s au sein d'une collaboration [4, 5]. Ils ne supportent pas ou ont des difficultés à supporter les activités de collaboration qui impliquent un nombre important de collaborateur·ice·s.

Confidentialité, Vie Privée, et Censure. Le rôle de carrefour du serveur permet la collecte massive de données privées et confidentielles, ainsi que le contrôle systématique de l'information et donc de son éventuelle censure [6].

Sécurité. Le rôle central du serveur en fait une cible privilégiée d'attaque [6]. La corruption du serveur permet à l'attaquant d'accéder à de nombreuses données sensibles des utilisateur·ice·s et éventuellement de les altérer.

Propriété des contenus. Le serveur détient les contenus partagés. Les collaborateur·ice·s perdent la propriété de leurs contenus et de leurs contributions. Le serveur peut ainsi restreindre la modification et la consultation des contenus produits. Il peut supprimer le contenu sans en alerter préalablement les contributeur·ice·s.

Modes de collaboration [7]. Les collaborateur·ice·s sont restreint·e·s dans les formes que peuvent prendre leur collaboration. Les infrastructures centralisées leur permettent de contribuer depuis des endroits géographiquement éloignés et à des instants différents. Toutefois elles et ils doivent être connecté·e·s au serveur. Ils ne peuvent pas travailler en sous-groupes, déconnecté·e·s les un·e·s des autres. La modification hors ligne et la consultation hors ligne ne sont pas toujours supportées.

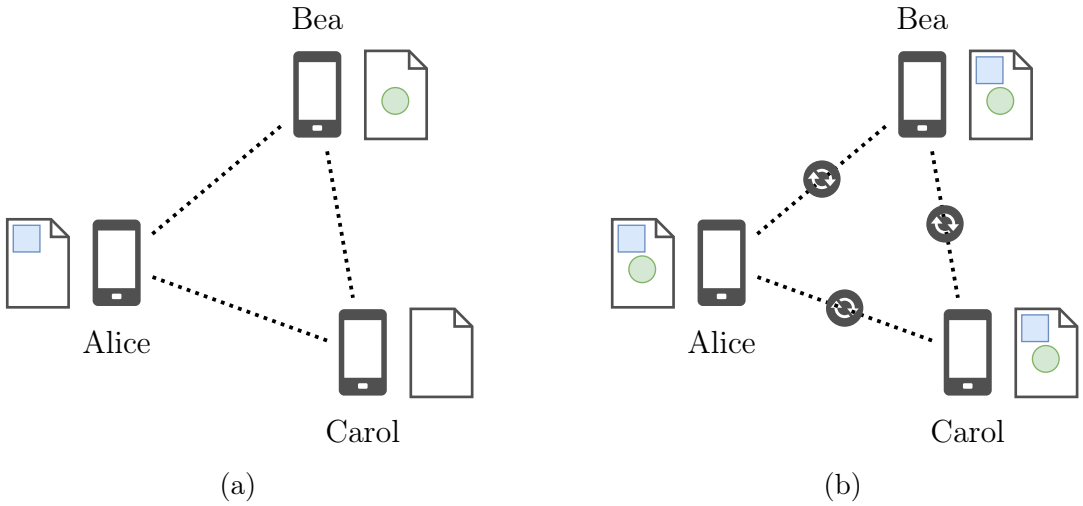


FIGURE 1.2 – Exemple d'une activité de collaboration au sein d'une infrastructure pair-à-pair. Alice, Bea, et Carol répliquent de manière optimiste un dessin. Elles disposent chacune d'une copie du dessin. (a) Alice et Bea modifient indépendamment leur copie du dessin. Alice ajoute un carré, alors que Bea ajoute un cercle. Les copies des collaboratrices divergent : les collaboratrices ont des dessins différents. (b) Elles synchronisent ensuite leurs copies afin d'intégrer les modifications de chacune. Les copies convergent : elles obtiennent ainsi le même dessin.

La diffusion des activités de collaboration a inspiré de nouveaux usages qui poussent les infrastructures centralisées à leurs limites. Par exemple, la prise de note en simultané durant un cours en ligne peut faire intervenir une promotion entière d'étudiant·e·s. Ces usages se sont renforcés avec la crise sanitaire provoquée par la pandémie de *COVID-19*.

Pour supporter ces nouveaux usages, plusieurs initiatives ont vu le jour [8, 9, 10]. Elles s'appuient sur des infrastructures pair-à-pair pour décentraliser les applications de collaboration. Les infrastructures pair-à-pair de collaboration sont caractérisées par l'absence de serveur central. Les appareils des collaborateur·ice·s sont des pairs. Ils sont directement connectés les uns aux autres et partagent les mêmes responsabilités [1].

Les infrastructures pair-à-pair de collaboration visent la conception d'applications de collaboration hautement disponibles, aux latences faibles, qui tolèrent les partitions réseaux, et passent à l'échelle [11]. Pour ce faire, elles reposent sur la réplication optimiste des contenus [12]. Chaque pair possède une copie des contenus partagés. Il interroge et modifie directement sa copie sans se coordonner avec les autres pairs. Les modifications sont propagées en arrière plan aux autres pairs. Les pairs intègrent à leur copie les modifications qu'ils reçoivent. La figure 1.2 illustre un scénario de réplication optimiste.

Les infrastructures pair-à-pair de collaboration donne plus de responsabilités aux pairs. Des collaborateur·ice·s peuvent davantage nuire au bon déroulement de la collaboration. Les infrastructures pair-à-pair permettent également la participation d'un plus grand nombre de collaborateur·ice·s. Ce qui nous amène à questionner la sécurité et le passage à l'échelle des protocoles de réplication optimiste.

1. Certaines infrastructures présentent des pairs privilégiés avec des responsabilités supplémentaires.

1.2 Problématiques et contributions

1.2.1 Convergence en présence de pairs malintentionnés

Les infrastructures pair-à-pair de collaboration permettent aux pairs de modifier en parallèle un contenu partagé. Pour ce faire, les pairs répliquent le contenu partagé et modifient leur copie respective. La modification concurrente du contenu engendre inévitablement la divergence des copies du contenu partagé [7]. Dans la figure 1.2, les copies divergent avant leur synchronisation. Les protocoles de réplication optimiste sont responsables de la réconciliation de l'état des copies pour les faire converger vers un état qui intègre les modifications de chacun. La convergence des copies conditionne la vivacité et donc le succès d'une collaboration. Si deux contributeur·ice·s ont des copies qui ne sont pas capables de converger, leur vue du contenu sera différente. Cette divergence de vue peut compromettre leur collaboration. Assurer à terme la convergence des copies est donc important dans une infrastructure de collaboration.

Pour illustrer l'importance de la convergence, nous prenons l'exemple de l'organisation d'un événement festif ou politique. Plusieurs réunions d'organisation ont lieu avant l'évènement. Pour garder une trace de ces réunions et informer les bénévoles absents, les organisatrices de l'événement ont mis en place un document textuel partagé. Les organisatrices et les bénévoles prennent collectivement des notes durant les réunions. Ce document peut par exemple inclure des dates de rendez-vous pour préparer l'événement. Si les pairs ne sont pas capables de converger, certains bénévoles risquent de manquer des rendez-vous essentiels. Ce qui pourrait retarder, voire empêcher, la tenue de l'événement.

Certains pairs peuvent trouver un intérêt à compromettre le succès d'une collaboration. Dans le cas de l'exemple précédent, il peut s'agir d'un opposant à la tenue de l'événement. Nous nous intéressons en particulier aux pairs qui entreprennent des actions pour faire diverger de manière permanente la vue des autres pairs. Ces pairs *malintentionnés* cherchent donc à faire diverger de manière permanente les copies des autres pairs que nous qualifions d'*honnêtes*. Pour ce faire, les pairs malintentionnés utilisent les faiblesses de certains protocoles de réplication : ils transmettent à différents pairs honnêtes des modifications distinctes qui sont perçues comme identiques par le protocole. Il s'agit d'une *équivoque*. Par exemple, l'opposant à l'événement peut infiltrer le groupe de bénévoles et faire en sorte que les bénévoles observent différentes dates de rendez-vous de préparation de l'événement. L'opposant répandrait ainsi la confusion et provoquerait l'absence de bénévoles à certains rendez-vous. La figure 1.3 illustre un scénario dans lequel un pair malintentionné effectue une équivoque. Les pairs honnêtes ne détectent pas l'équivoque. Leurs copies divergent de manière permanente.

Ce qui nous amène à formuler notre première question de recherche : **Est-il possible d'assurer la convergence des copies des pairs honnêtes en présence de pairs malintentionnés et de conserver les propriétés offertes par les infrastructures pair-à-pair de collaboration ?**

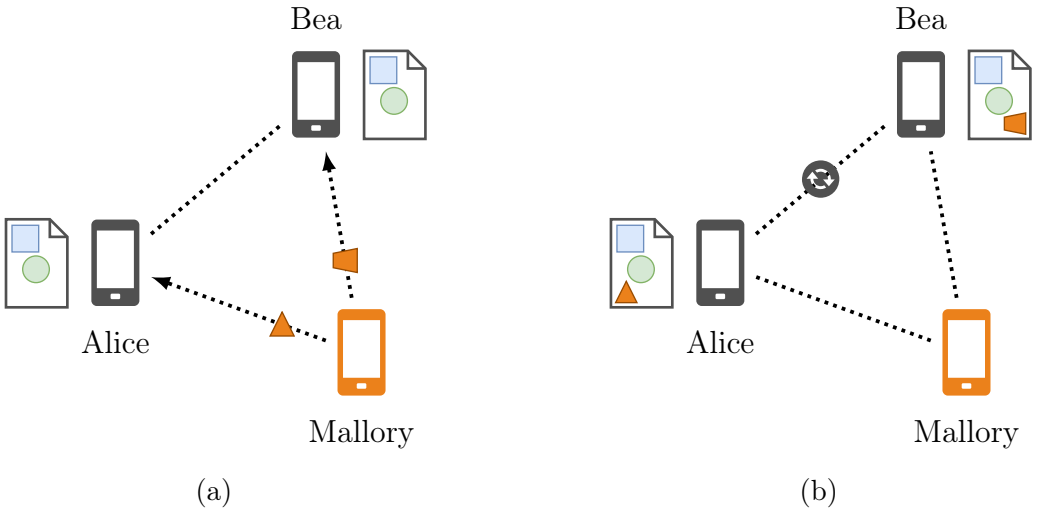


FIGURE 1.3 – Exemple d'une équivoque qui vise à faire diverger de manière permanente les copies des collaboratrices honnêtes. Alice et Bea sont honnêtes. Elles répliquent de manière optimiste un dessin. Mallory est malintentionnée. Elle souhaite compromettre la collaboration entre Alice et Bea. (a) Mallory effectue une équivoque. Elle transmet à Alice l'ajout d'un triangle orange, alors qu'elle transmet à Bea l'ajout d'un trapèze orange. Elle fait en sorte que ces modifications distinctes paraissent identiques. Par exemple, si l'on considère que la couleur est la seule information utilisée pour détecter la présence d'une forme sur le dessin, alors il n'est pas possible de différencier un dessin avec la première forme d'un dessin avec la seconde. (b) Alice et Bea intègrent la modification qu'elles reçoivent. Leurs copies divergent. Elles se synchronisent ensuite, mais elles ne détectent pas la divergence de leur copie. L'objectif de Mallory est atteint : les copies de Alice et Bea demeurent divergentes.

De nombreux protocoles de réplication optimiste synchronisent les copies en échangeant des opérations [13]. L'opération synthétise une modification effectuée sur l'une des copies. Un pair malintentionné met en œuvre une équivoque en générant deux opérations distinctes mais qui sont perçues comme identiques par le protocole de réplication. Pour détecter de telles opérations, les pairs peuvent utiliser des *journaux infalsifiables* d'opérations [14, 15, 16]. Chaque pair possède un journal dans lequel il enregistre les opérations qu'il transmet et qu'il reçoit. Chaque opération est signée par son auteur et référence des opérations sur lesquelles elle dépend. Deux opérations distinctes ont une signature différente. Les pairs peuvent donc identifier les équivoques. En considérant que le protocole de réplication puisse intégrer des opérations distinctes présentées comme identiques, les journaux infalsifiables permettent donc d'assurer la convergence des copies des pairs honnêtes.

L'utilisation de journaux infalsifiables représente un coût supplémentaire pour la collaboration. Les pairs doivent conserver le journal, et donc l'ensemble des opérations qu'ils ont transmis et reçu. Ce coût est en particulier visible lorsqu'un nouveau pair rejoint la collaboration. Le nouveau pair doit en effet récupérer l'intégralité du journal pour obtenir l'état actuel du contenu partagé. Au fur et à mesure de la collaboration, le journal croît. Ce qui rend coûteux la transmission du journal et la construction de l'état actuel du contenu partagé à partir du journal. L'utilisation de journaux infalsifiables passe donc difficilement

à l'échelle. Ils ne permettent pas de tirer pleinement avantage des protocoles de réplication optimiste et des infrastructures pair-à-pair.

Certains protocoles de réplication [17, 18] permettent la transmission de l'état d'une copie aux nouveaux pairs. L'état d'une copie occupe généralement un espace mémoire plus faible que le journal d'opérations. Le coût de transmission est ainsi plus faible et les nouveaux pairs obtiennent directement le contenu partagé. Malheureusement les pairs malintentionnés peuvent compromettre la convergence des copies des pairs honnêtes en transmettant des états falsifiés. Ils peuvent par exemple transmettre un ancien état et déclarer qu'il est à jour ou ajouter des modifications qui n'ont pas été transmises aux autres pairs.

Pour réduire le coût lié à l'utilisation de journaux infalsifiables, nous proposons un protocole qui permet de tronquer un journal infalsifiable et d'authentifier un état à l'aide d'un journal tronqué. La troncature d'un journal infalsifiable est réalisée de telle manière à ne pas compromettre la détection des équivoques des pairs malintentionnés. Pour ce faire, elle repose sur le concept de *stabilité*. Une opération est stable dans un journal si toute opération acceptée ultérieurement dans le journal dépend sur cette opération. Pour rejoindre une collaboration, les nouveaux pairs récupèrent un état et le journal tronqué qui lui est associé. Le protocole leur permet de rejeter les états falsifiés par des pairs malintentionnés.

1.2.2 Protocole de réplication pour la co-édition de texte

L'édition collaborative de texte permet à des collaborateur·ice·s géographiquement éloigné·e·s de co-rédiger un texte simultanément ou de manière différée dans le temps. Les collaborateur·ice·s doivent pouvoir consulter et modifier à tout moment le document textuel. En simultané, les latences doivent être suffisamment faibles pour éviter les conflits de modification et avoir conscience de l'activités des autres collaborateur·ice·s. Ces dernières années, l'usage de l'édition collaborative s'est étendue à des activités de collaboration qui impliquent de nombreux rédacteur·ice·s, telle que la prise de note pour un cours en ligne. L'édition collaborative de texte présente donc un défi car elle requiert à la fois une haute disponibilité, des latences basses, et un passage à l'échelle raisonnable.

L'usage de l'édition collaborative s'est répandu grâce à des éditeurs de texte collaboratifs tel que *EtherPad*[2], *Google Docs*[3], et *ShareLaTeX / Overleaf*[4]. Ces éditeurs collaboratifs reposent sur des infrastructures centralisées. Ils permettent à une dizaine d'utilisateur·ice·s d'éditer en simultané un document textuel. En revanche lorsque ce seuil est dépassé, les utilisateur·ice·s remarquent les latences [19]. Elles et ils perçoivent les délais entre le moment où les modifications sont effectuées et le moment où elles sont intégrées. Elles et ils adoptent des stratégies pour limiter les conflits de modification [4, 5]. Ce qui les conduit à réduire leurs interactions et donc à réduire l'efficacité de leur collaboration. Au-delà de dix utilisateur·ice·s, *Etherpad* rejette des connexions. *Google Docs* expose le même comportement lorsqu'une quarantaine d'utilisateur·ice·s sont actifs [19].

2. etherpad.org
3. docs.google.com
4. www.overleaf.com

Pour dépasser ces limitations, la communauté scientifique a proposé plusieurs protocoles de réplication optimiste qui tirent avantage des infrastructures pair-à-pair [20, 21, 22, 23, 24, 25, 26, 27]. Dans ces protocoles, les pairs échangent leurs modifications sous la forme d'opérations pour synchroniser leurs copies. Une opération insère ou supprime un caractère dans le document. Les protocoles s'assurent que chaque opération est intégrée exactement une fois. Ils s'assurent également que certaines opérations sont intégrées avant d'autres. Par exemple l'intégration de la suppression d'un caractère doit survenir après l'intégration de l'insertion de ce caractère. Selon le protocole, il peut exister d'autres ordres d'intégration à respecter. Le respect de l'ordre d'intégration est souvent assuré par une couche de livraison d'opérations. En pratique, cette couche de livraison ne prend pas en compte les ordres spécifiques d'intégration de chaque protocole de réplication [28, 29]. Une livraison causale [30] des opérations est généralement utilisée : si un pair intègre un ensemble d'opérations, alors cet ensemble doit être intégré sur toute copie avant l'intégration des opérations ultérieures de ce pair. La figure 1.4 donne un exemple de livraison causale et par extension d'intégration causale d'opérations lors d'une session collaborative.

L'intégration des opérations dans un ordre spécifique met potentiellement en attente des intégrations. En effet, l'intégration d'une opération est retardée jusqu'à ce que les opérations qui doivent être préalablement intégrées le soient. La perte d'opérations sur le réseau ou la déconnexion de pairs peut donc propager des ralentissements dans l'ensemble du système [31].

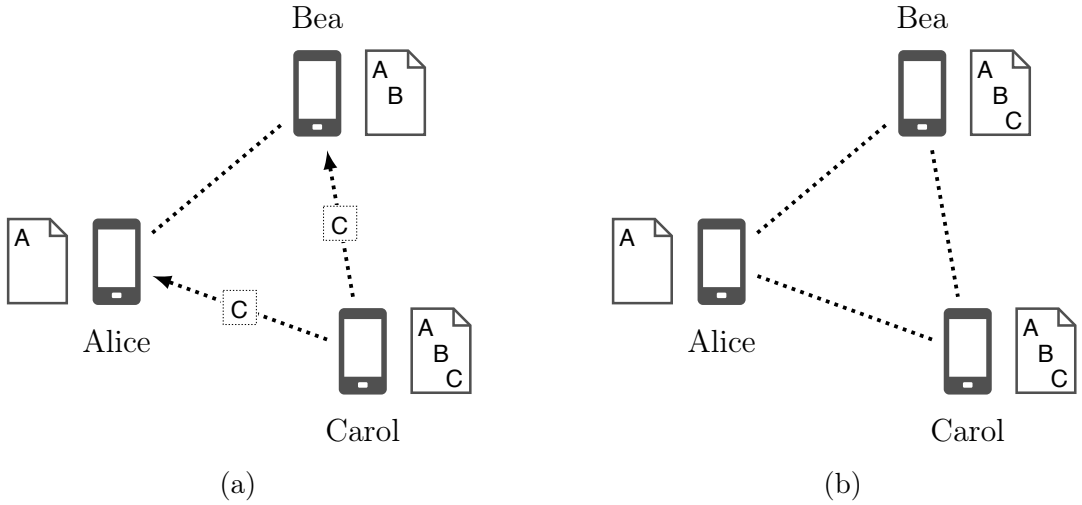


FIGURE 1.4 – Exemple d'intégration causale d'une opération. Un ensemble de pairs réplique un texte. La synchronisation des copies reposent sur une livraison causale des modifications. (a) Carol insère le caractère 'C' à sa copie qui a déjà intégré les caractères 'A' et 'B'. Elle propage sa modification a Alice et Bea. Pour intégrer le caractère 'A' sur une copie, la copie doit déjà contenir les caractères 'A' et 'B'. Alice n'a pas reçu l'insertion du caractère 'B'. Elle ne peut donc pas insérer le caractère 'C'. Elle retarde l'intégration de cette modification jusqu'à la réception du caractère 'B'. En revanche, la copie de Bea contient déjà les caractères 'A' et 'B'. (b) Elle intègre donc la modification de Carol.

Généralement, ces protocoles supportent par conception l'édition hors-ligne et en sous-groupes déconnectés les uns des autres. Cependant, lorsque les périodes de déconnexion sont longues ou le nombre de modifications hors ligne est important, la synchronisation est coûteuse en communication et en ressources de calcul. En effet, chaque opération est transmise et intégrée aux copies des pairs.

Ce qui nous amène à la deuxième question de recherche que nous explorons dans ce manuscrit : **Est-il possible de concevoir un protocole de réplication optimiste pour l'édition collaborative de texte en simultané qui intègre immédiatement toute modification et est adapté aux longues périodes de déconnexion ?**

A notre connaissance, la synchronisation par opérations est le seul schéma de synchronisation utilisé pour l'édition collaborative de texte en simultané. D'autres schémas de synchronisation ont été proposés pour la conception de protocole de réplication. La synchronisation par états [28] consiste à transmettre l'état de la copie une fois mise à jour au lieu d'une opération. Les états sont fusionnés de telle sorte à inclure les mises-à-jours de chacun. Ce schéma tire son avantage principal du fait qu'un état synthétise l'ensemble des modifications qui ont été précédemment intégrées. Les états peuvent donc être livrés et intégrés plusieurs fois dans des ordres arbitraires. Ce qui implique qu'ils peuvent être dupliqués, omis, et réordonnés par le réseau sans compromettre la convergence des copies. Les pairs peuvent donc intégrer un état dès sa réception. Il n'y a pas d'intégration retardée d'états. Par ailleurs, le système est moins sensible aux longues périodes de déconnexions des pairs. Au lieu de transmettre une longue liste d'opérations, les pairs transmettent simplement l'état de leur copie.

Malgré ces avantages, le schéma de synchronisation par états est resté circonscrit à un petit nombre de protocoles de réplication. Le schéma implique en effet un coût en communication et en fusion important lorsque les états sont de taille importante. C'est le cas des copies de contenu textuel. En effet, il n'est pas raisonnable de transmettre et intégrer l'état d'un document chaque fois qu'un caractère est inséré ou supprimé du document.

Un schéma de synchronisation par différences d'états [18] a récemment été proposé pour tenter de réduire le coût du schéma de synchronisation par états. Ce schéma autorise toujours la transmission d'états complets. Cependant, il privilégie l'envoi de différences d'états qui synthétisent une ou plusieurs modifications. Les différences d'états peuvent être intégrées plusieurs fois dans des ordres arbitraires. Le schéma de synchronisation par différences d'états offre ainsi de nombreuses possibilités dans la manière de synchroniser deux copies. Par exemple, lorsque des pairs collaborent en simultané, ils peuvent échanger des différences d'états, alors que deux pairs qui se reconnectent après une longue période de déconnexion peuvent échanger directement leur état.

Nous proposons d'adopter ce schéma de synchronisation pour l'édition collaborative de texte. Pour ce faire, nous concevons un nouveau protocole de réplication optimiste nommé *Dotted LogootSplit*.

1.3 Organisation du manuscrit

Le reste du manuscrit est organisé en 5 chapitres :

Le chapitre 2 introduit plus en détail les caractéristiques des infrastructures pair-à-pair de collaboration et met l'accent sur les problématiques de recherche liées à la convergence des copies et à sa protection. Le chapitre définit également l'adversaire du système qui contrôle les pairs malintentionnés.

Le chapitre 3 présente les formalismes que nous utilisons dans les chapitres suivants pour décrire nos contributions. Nous introduisons en particulier la notion de cohérence de données et les types de données répliqués.

Le chapitre 4 présente deux protocoles qui protègent la convergence des copies des pairs honnêtes. Le protocole à journaux complets protège la convergence des copies des pairs honnêtes à l'aide de journaux infalsifiables et complets. Le protocole à journaux tronqués repose sur le protocole à journaux complets. Il permet la troncature des journaux et l'authentification de l'état d'une copie à l'aide d'un journal tronqué. Pour développer ce protocole, le chapitre introduit la notion de stabilité et *DynVFJC*, un nouveau modèle de cohérence. Nous présentons également les travaux de la littérature en relation avec notre protocole.

Le chapitre 5 présente un état de l'art des types séquences de données répliquées. Nous proposons un modèle unifié pour décrire et mieux appréhender un sous-ensemble de ces types séquences. À l'aide de ce modèle, nous répondons à notre deuxième question de recherche en proposant *Dotted LogootSplit*, le premier type séquence de données répliquées synchronisé par différences d'états.

Le chapitre 6 clôt ce manuscrit et propose des axes d'exploration pour continuer les travaux de recherche présentés au sein de ce manuscrit.

1.4 Publications

La première contribution développée au sein de ce manuscrit a fait l'objet d'une publication [32]. La seconde contribution présentée n'a pas encore fait l'objet d'une publication. En revanche, la proposition a été implémentée dans notre prototype d'éditeur collaboratif *MUTE* qui est décrit dans l'une de nos publications [33]. Notre dernière publication [34] est en lien avec nos travaux sur les types de données répliquées, mais n'est pas présentée au sein de ce manuscrit. Dans la suite de cette section, nous référençons les articles qui ont été publiés au cours de cette thèse.

Prunable Authenticated Log and Authenticable Snapshot in Distributed Collaborative Systems [32]

– Victorien Elvinger, Gérald Oster, François Charoy

Article de conférence in proceedings of the 4th IEEE International Conference on Collaboration and Internet Computing, (CIC 2018)

Abstract: In distributed collaborative systems, participants maintain a replicated copy of shared documents. They edit their own copy and then share their modifications without any coordination. Copies follow successions of divergence and convergence. Convergence is a liveness property of collaborative systems. Some malicious participants may find an advantage to make the collaboration fail. To that end, they can preclude convergence of the copies. To protect convergence of copies, participants can exploit an authenticated log of modifications. New participants have to retrieve the entire log in order to contribute. Unfortunately, the cost of joining a collaboration increases with the size of this log. Causal Stability allows to prune authenticated logs in a static collaborative group without any malicious participants. In this paper, we first tailor Causal Stability to dynamic groups in presence of malicious participants. We then propose a mechanism to verify the consistency of a pruned log and a mechanism to authenticate a snapshot from a pruned log.

MUTE: A Peer-to-Peer Web-based Real-time Collaborative Editor [33]

– Matthieu Nicolas, Victorien Elvinger, Gérald Oster, Claudia-Lavinia Ignat, François Charoy

Article de démonstration in proceedings of the 15th European Conference on Computer Supported Cooperative Work (ECSCW 2017)

Abstract: Real-time collaborative editing allows multiple users to edit shared documents at the same time from different places. Existing real-time collaborative editors rely on a central authority that stores user data which is a perceived privacy threat. In this paper, we present Multi-User Text Editor (MUTE), a peer-to-peer web-based real-time collaborative editor without central authority disadvantages. Users share their data with the collaborators they trust without having to store their data on a central place. MUTE features high scalability and supports offline and ad-hoc collaboration.

A Generic Undo Support for State-Based CRDTs [34]

– Weihai Yu, Victorien Elvinger, Claudia-Lavinia Ignat

Article de conférence in proceedings of the 23rd International Conference on Principles of Distributed Systems, (OPODIS 2019)

Abstract: CRDTs (Conflict-free Replicated Data Types) have properties desirable for large-scale distributed systems with variable network latency or transient partitions. With CRDT, data are always available for local updates and data states converge when the replicas have incorporated the same updates. Undo is useful for correcting human mistakes and for restoring system-wide invariant violated due to long delays or network partitions. There is currently no generally applicable undo support for CRDTs. There are at least two reasons for this. First, there is currently no abstraction that we can practically use to capture the relations between undo and normal operations with respect to concurrency and causality. Second, using inverse operations as the existing partial solutions, the CRDT designer has to hard-code certain rules and design a new CRDT for almost every operation that needs undo support. In this paper, we present an approach to generic support of undo for CRDTs. The approach consists of two major parts. We first work out an abstraction that captures the semantics of concurrent undo and redo operations through equivalence classes. The abstraction is a natural extension of undo and redo in sequential applications and is straightforward to implement in practice. By using this abstraction, we then device a mechanism to augment existing CRDTs. The mechanism provides an "out of the box" support for undo without the involvement of the CRDT designers. We also present a practical application of the approach in collaborative editing.

Chapitre 2

Problématiques

Sommaire

Les infrastructures pair-à-pair de collaboration peuvent supporter les activités massives de collaboration. Elles tolèrent les latences, les pannes, ainsi que les partitions réseaux. Les collaborateur·ice·s peuvent modifier le contenu partagé de manière isolée. Des collaborateur·ice·s géographiquement éloigné·e·s ou avec des connexions réseaux lentes et peu fiables peuvent contribuer à une collaboration. Les collaborateur·ice·s ont donc l'illusion d'observer les modifications en temps réel lorsqu'elles et ils modifient simultanément un contenu. Ces caractéristiques permettent de passer à l'échelle et de respecter le rythme de vie, ainsi que la manière de collaborer de chaque collaborateur·ice.

Pour ce faire, les infrastructures pair-à-pair de collaboration reposent sur la *réplication de données* [12]. Chaque collaborateur·ice est associé·e à un pair qui possède une copie du contenu partagé. Les algorithmes de réplication de données sont responsables de la convergence des copies. La convergence des copies conditionne le succès d'une collaboration. Si les collaborateur·ice·s ne sont pas capables de partager une vue globalement convergente dans des délais raisonnables, leur collaboration est probablement compromise. Assurer à terme la convergence des copies est donc important.

Les algorithmes existants de réplication de données supposent généralement l'absence de collaborateur·ice·s malintentionné·e·s. Plus largement, ils supposent l'absence d'un adversaire actif. Cette hypothèse n'est pas réaliste au sein d'infrastructures pair-à-pair. Les pairs partagent les mêmes responsabilités et peuvent nuire au succès de la collaboration. Certains pairs peuvent par exemple mener des attaques pour assurer une divergence permanente des copies des autres pairs. Dans ce manuscrit de thèse, nous nous intéressons à cette problématique et nous proposons des mécanismes de défense pour assurer à terme la convergence.

Ce chapitre introduit dans un premier temps la réplication de données. Il développe ensuite la notion de convergence. Il termine sur le modèle du système auquel nous nous intéressons et il définit l'adversaire du système.

2.1 Réplication de données

La réplication de données maintient des copies de contenus partagés sur un ensemble de pairs. Elle augmente la disponibilité d'un contenu en permettant son interrogation et sa modification en dépit de la défaillance de pairs. Les répliques sont souvent géographiquement réparties. Ce qui rapproche les données des utilisateur·ice·s. Elle réduit donc les latences pour interroger les contenus partagés.

Comparé à un contenu non-répliqué, un contenu répliqué a un nombre étendu de comportements. Par exemple, des collaborateur·ice·s distinct·e·s peuvent modifier des copies distinctes d'un contenu répliqué. Ces nouveaux comportements peuvent engendrer des scénarios difficiles à prévoir et traiter. Idéalement, l'interrogation et la modification d'un contenu répliqué devrait se comporter comme s'il existait une seule copie. Dans ce cas, nous disons que le contenu partagé est *fortement cohérent* et nous parlons de réplication pessimiste de données [12]. La figure 2.1 met en évidence que l'interrogation d'un contenu fortement cohérent prend toujours en compte toutes les modifications précédemment effectuées.

Le maintien de l'illusion de l'interrogation et de la modification d'une seule copie a un coût. Elle requiert une coordination étroite entre les pairs. Une coordination qui peut se traduire par des communications systématiques avant chaque modification, voire avant chaque interrogation. Les latences et les défaillances inhérentes aux réseaux [35] rendent cette coordination problématique. Elle engendre plusieurs problèmes :

Disponibilité. Le contenu ne peut plus être modifié, voire interrogé, lorsque des pairs sont isolés par des partitions réseaux.

Latence. L'interrogation et la modification du contenu partagé est sujet aux latences du réseau et des pairs. La lenteur d'un pair ou d'une connexion réseau peut donc affecter l'ensemble du système.

Passage à l'échelle. L'ajout de pairs, et donc de copies, peut détériorer la disponibilité et les latences, étant donné que le nombre de pairs impliqués dans une coordination augmente.

Le théorème CAP [36, 37] montre l'impossibilité de garantir à la fois une cohérence forte et une disponibilité à tout moment en présence de partitions réseaux. Nous pouvons illustrer cette impossibilité en prenant l'exemple d'une bibliothèque municipale qui possède un système répliqué d'emprunts de livres. Deux usagers souhaitent faire un emprunt. Les usagers sont connecté·e·s à des pairs distincts. Supposons que le réseau est momentanément partitionné de manière à ce que les pairs considérés se trouvent dans l'impossibilité de communiquer. Le système doit faire le choix entre le maintien d'une cohérence forte des emprunts ou la disponibilité d'emprunter. Le maintien de la cohérence forte empêche tout emprunt tant que les partitions réseaux persistent. La disponibilité d'emprunter en

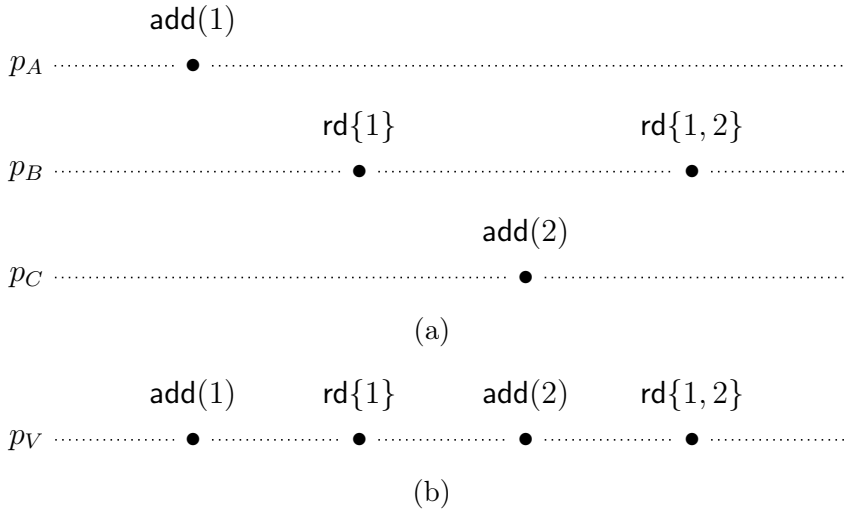


FIGURE 2.1 – (a) Réplication pessimiste d'un ensemble. L'ensemble répliqué supporte une opération de modification $\{\mathsf{add}(n) \mid n \in \mathbb{N}_0\}$ qui ajoute un entier dans l'ensemble, et une opération d'interrogation rd qui donne la composition de l'ensemble. Ainsi l'exécution de l'opération $\mathsf{add}(1)$ ajoute l'entier 1 à l'ensemble, et nous notons $\mathsf{rd}\{1\}$ une opération de lecture suivie de la valeur retournée par son exécution. L'ensemble est initialement vide. Le pair p_A ajoute l'entier 1. Ultérieurement le pair p_C ajoute l'entier 2. Le pair p_B interroge à deux reprises l'ensemble. Sa première interrogation se déroule entre les deux ajouts. La seconde interrogation se déroule après les deux ajouts. L'ensemble est fortement cohérent, une modification qui survient avant une autre modification ou une interrogation est donc visible à cette dernière. La première interrogation retourne un singleton composé de l'élément 1 et la seconde interrogation retourne un ensemble composé des éléments 1 et 2. La cohérence forte maintient l'illusion de l'interrogation et la modification d'une seule copie. (b) L'histoire de l'exécution de cette copie abstraite peut être obtenue en projetant toutes les opérations sur un pair abstrait p_V.

dépit du partitionnement peut conduire à des conflits. Par exemple, des usagers peuvent effectuer des emprunts sur la même période d'un livre disponible en un seul exemplaire.

L'illustration proposée pourrait nous amener à penser que la cohérence forte peut être difficilement sacrifiée au bénéfice de la disponibilité. En effet, le risque d'emprunts conflictuels pourrait produire des situations désagréables pour les usagers et la bibliothèque. L'illustration proposée souffre du problème de la *double-dépense* [38]. L'emprunt d'un exemplaire d'un livre sur une période donnée correspond à une ressource consommable. Cette ressource ne peut pas être consommée (dépensée) deux fois. Ce problème n'est pas toujours rencontré. Par exemple, un système de pétition en ligne ne souffre pas du problème de la *double-dépense*. Plusieurs personnes peuvent signer une pétition sans avoir besoin de se coordonner.

Les applications n'ont pas toujours besoin de maintenir une cohérence forte. En général, le maintien d'une cohérence plus faible est suffisant [39, 40, 41]. Prenant cette observation en considération, la **réplication optimiste** [12] sacrifie la cohérence forte pour atteindre une haute disponibilité et des latences basses. Elle autorise la modification et l'interrogation

des copies sans aucune coordination. Les pairs modifient indépendamment leur copie. Ce qui conduit à la divergence des copies. Deux copies divergentes fournissent des réponses différentes à au moins une interrogation. Les pairs se synchronisent pour échanger leurs modifications. Ils intègrent les modifications de chacun à leur copie. Les modifications sont donc intégrées dans des ordres distincts. Les protocoles de réplication optimiste sont responsables de la convergence des copies. La figure 2.2 illustre un scénario de réplication optimiste.

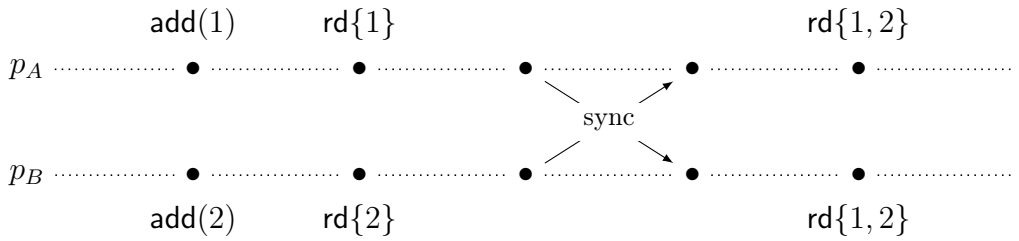


FIGURE 2.2 – Exemple de réplication optimiste d'un ensemble. Les pairs p_A et p_B débutent avec un ensemble vide. p_A ajoute l'élément 1 sur sa copie. p_B ajoute en parallèle l'élément 2 sur sa copie. Les copies divergent : les deux pairs obtiennent des singletons distincts. Ils synchronisent ultérieurement leurs copies pour échanger et intégrer leurs modifications respectives. Les modifications ne sont pas intégrées dans le même ordre sur les copies : ils intègrent la modification de l'autre pair après leur propre modification. Les copies finissent par converger : les pairs obtiennent le même ensemble.

2.2 Convergence des copies

Un protocole de réplication optimiste peut garantir plusieurs modèles de cohérence dits « faibles ». L'une des garanties élémentaires de ces modèle concerne la convergence des copies. En l'absence de modifications, les copies devraient converger. Toute interrogation sur deux copies convergentes donne la même réponse. La convergence peut être définie de différentes manières. La *cohérence à terme* [42, 12, 43] offre la garantie de convergence la plus faible qu'un protocole de réplication optimiste doit respecter. Elle garantit que deux copies finissent par converger dès lors qu'elles ont intégré le même ensemble de modifications. Elle garantit également qu'une modification intégrée par un pair finit par l'être par les autres pairs.

Definition 2.1 (Cohérence à terme)**.** Une exécution qui respecte la cohérence à terme, respecte les propriétés suivantes :

EC1 (intégration à terme) Une modification intégrée sur la copie d'un pair est à terme intégrée par l'ensemble des pairs.

EC2 (convergence à terme) Les pairs qui ont intégré le même ensemble de modifications ont à terme, des copies convergentes.

La convergence des copies est généralement une tâche difficile. Cette difficulté réside principalement dans la présence de conflits de modification. La figure 2.3 présente un scénario où un conflit de modification survient. La réplication optimiste fait l'hypothèse « optimiste » que les conflits de modification sont rares et peuvent être résolus ultérieurement. Les pairs doivent donc gérer les conflits de modification et se mettre d'accord sur un moyen de les résoudre.

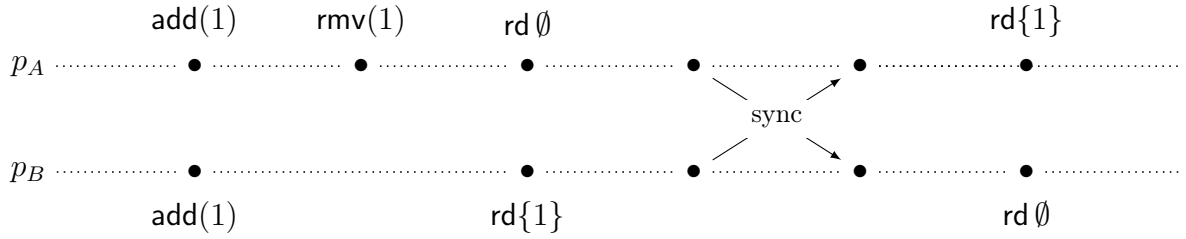


FIGURE 2.3 – Exemple de conflit de modification. Les pairs p_A et p_B répliquent un ensemble qui supporte une opération d'ajout $\{\mathsf{add}(n) \mid n \in \mathbb{N}_0\}$, une opération de suppression $\{\mathsf{rmv}(n) \mid n \in \mathbb{N}_0\}$, et une opération de lecture rd. p_A ajoute l'élément 1 puis le supprime. En parallèle, p_B ajoute également l'élément 1. L'ajout et la suppression en parallèle d'un même élément constitue un conflit de modification. L'élément ne peut pas être à la fois présent et absent de l'ensemble. L'intégration « naïve » des modifications dans des ordres distincts conduit à une divergence des copies.

Le succès d'une collaboration est conditionné par la capacité des copies à converger. IGNAT et al. [4] ont montré que les activités de collaboration en simultané étaient compromises lorsque le délai entre des modifications de contenu et leurs observations par les autres était trop important. Dans l'idéal, la convergence devrait être atteinte le plus rapidement possible. Pour réduire au maximum les délais de convergence, toute coordination devrait être évitée [44].

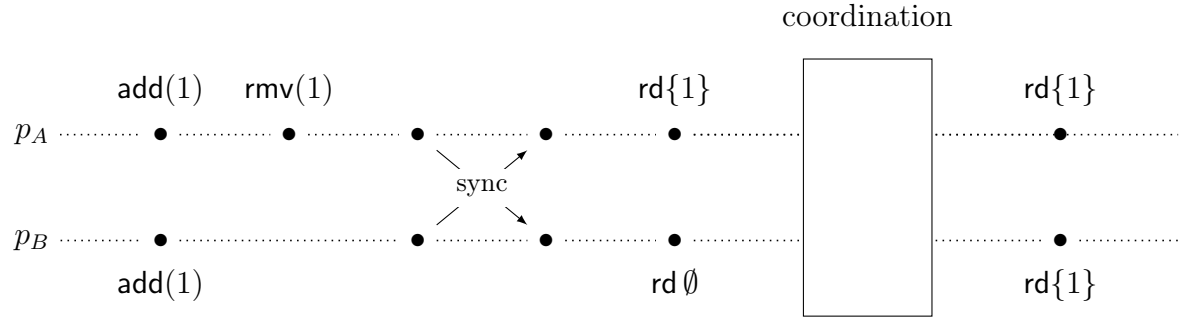


FIGURE 2.4 – Exemple de coordination après l'intégration de modifications. Les pairs p_A et p_B intègrent les modifications de chacun. Les copies ne convergent pas. Les pairs se coordonnent après l'intégration des modifications afin de converger.

Un protocole de réplication optimiste garantit une *convergence à terme* des copies des pairs. La convergence à terme assure seulement que les pairs qui ont intégré un même ensemble de modifications convergeront dans un futur plus ou moins proche. Elle ne

donne aucune borne sur le temps nécessaire et le nombre de communications nécessaires pour converger. La figure 2.4 illustre un scénario dans lequel deux pairs intègrent leurs modifications respectives et se coordonnent ensuite afin de converger.

Cette propriété de convergence est faible car il s'agit d'une propriété de vivacité. Une propriété de vivacité peut être violée en plusieurs points de l'exécution d'un protocole [45]. Elle décrit « quelque chose de positif » qui devrait survenir. La formulation d'une propriété de vivacité inclut généralement l'expression « à terme ».

Afin de faire converger plus rapidement les copies et exclure l'utilisation de coordinations, nous devons contraindre la convergence avec des propriétés de sûreté. Une propriété de sûreté décrit « quelque chose de mauvais » qui ne devrait jamais survenir. Une propriété de sûreté doit être respectée en tout point de l'exécution d'un protocole. Shapiro et al. [28] proposent la *convergence forte* comme propriété de sûreté de la convergence.

Definition 2.2 (Convergence forte)**.** [28] Les pairs qui ont intégré le même ensemble de modifications ont des copies convergentes.

La *convergence forte* ne nécessite pas de coordination après l'intégration des modifications. Cependant, elle n'exclut pas l'utilisation de coordination avant l'intégration des modifications. Les pairs peuvent décider d'un moyen de résoudre les éventuels conflits de modification. Par exemple, un protocole peut utiliser un pair privilégié qui ordonne les modifications selon un ordre global [46]. Mahajan et al. [47] proposent la *convergence unidirectionnelle* pour éviter toute coordination. L'idée centrale de la *convergence unidirectionnelle* est que deux pairs p_A et p_B devraient être capables de faire converger leur copie en deux étapes de communication unidirectionnelle : p_A envoie des modifications à p_B, puis p_B envoie des modifications à p_A.

La non-nécessité de coordination pour obtenir des copies convergentes suggère une résolution déterministe des conflits de modification. Dans le cas de la bibliothèque, nous pouvons supposer que dans la majorité des cas, des emprunts simultanés concernent des livres ou des exemplaires différents sur des périodes distinctes. Toutefois, des conflits d'emprunts peuvent survenir. Le nombre d'exemplaires d'un nouveau livre peut être insuffisant par rapport au nombre de lecteur·ice·s intéressé·e·s. Plusieurs stratégies peuvent être mises en place pour résoudre ces conflits. On peut considérer que l'emprunteur·ice final·e correspond à la personne qui a fait l'emprunt en premier.

2.3 Modèle du système

Dans les infrastructures pair-à-pair de collaboration, chaque collaborateur·ice est associé·e à un pair qui lui est exclusif. Guerraoui et al. [48] parlent de *sticky availability*. L'une des principales implications est que des pairs peuvent joindre et quitter le système de réplication au gré des connexions et des déconnexions des collaborateur·ice·s. De nouveaux pairs peuvent joindre lorsque de nouveaux et nouvelles collaborateur·ice·s joignent le groupe Des pairs existants peuvent se déconnecter indéfiniment lorsque les collaborateur·ice·s associé·e·s quittent le groupe.

Pour interroger et modifier le contenu partagé, un·e collaborateur·ice soumet des opérations au pair qui lui est associé. Ces opérations dépendent du type du contenu répli-

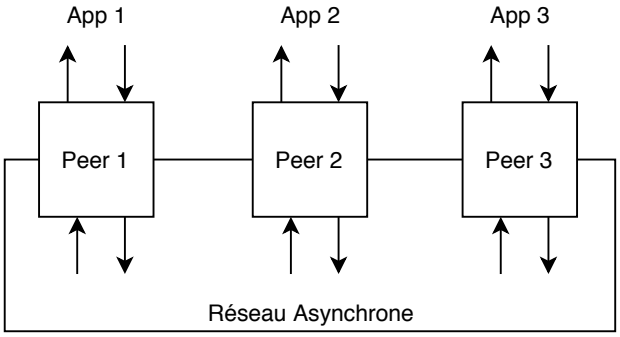


FIGURE 2.5 – Une infrastructure de collaboration composée de trois pairs qui répliquent le contenu partagé. Les trois pairs sont connectés par un réseau asynchrone.

qué. Dans la figure 2.2, l'ensemble répliqué peut être modifié par une opération d'ajout $\{\mathsf{add}(n) \mid n \in \mathbb{N}_0\}$ et peut être intérrogé par une opération de lecture rd qui donne la composition de l'ensemble. Les opérations sont d'abord intégrées sur la copie que le pair détient. Les pairs synchronisent leur copie en arrière-plan. Une synchronisation peut s'effectuer après chaque nouvelle modification ou peut être espacée dans le temps.

Nous supposons qu'un·e collaborateur·ice ne peut être dissocié·e du pair qui lui est attaché. La copie est donc *toujours disponible* [47] pour être interrogée et modifiée.

2.4 Adversaire du système

Un protocole de réplication optimiste doit respecter un ensemble de propriétés de cohérence. Pour assurer ces propriétés, un protocole doit résister aux défaillances qu'il rencontre au cours de ses exécutions. Par exemple, il doit se prémunir contre les défaillances inhérentes aux réseaux.

Les protocoles de réplication optimiste supposent généralement l'absence de comportements malintentionnés. Cette hypothèse n'est pas réaliste au sein d'activités massives de collaboration. Plus un groupe de collaborateur·ice·s est large, plus il est probable que le groupe inclue un ou plusieurs collaborateur·ice·s malintentionné·e·s.

Les défaillances d'un système peuvent être accidentelles ou résultantes de comportements malintentionnés. Sans perte de généralité, nous considérons que toute défaillance est le résultat de comportements malintentionnés. Ces comportements sont exprimés par l'adversaire du système. L'adversaire du système est mû par un objectif et contrôle des ressources telles que le réseau et des pairs, pour atteindre son objectif.

L'objectif de notre adversaire est de compromettre la convergence des copies des collaborateur·ice·s honnêtes [5]. Et ce dans le but de faire échouer leur collaboration. Pour simplifier les références ultérieures, nous proposons de classifier les capacités de l'adversaire à travers trois profils : *réseau asynchrone non-fiable*, *réseau corrompu*, *pairs perturbés*, et *pairs malintentionnés*.

5. Dans la littérature le terme « correct » est également utilisé.

Réseau asynchrone non-fiable. L'adversaire contrôle le réseau et peut exposer l'ensemble des comportements observés sur un réseau asynchrone non-fiable [49]. En particulier, les messages échangés entre les pairs peuvent être réordonnés, dupliqués, retardés, et omis. L'adversaire peut séparer les pairs en partitions. Toutefois il ne peut pas empêcher les pairs honnêtes de s'échanger à terme un nombre non-borné de messages. Ce qui implique qu'il ne peut pas maintenir indéfiniment une partition réseau ou omettre tous les messages envoyés par un pair. Cette limitation est nécessaire pour que des propriétés de vivacité, telles que l'*intégration à terme*, soient réalisables.

Réseau asynchrone corrompu. L'adversaire a les limitations et les capacités décrites dans le profil *réseau asynchrone non-fiable*. En outre, il peut contrefaire des messages. Il peut par exemple changer l'émetteur d'un message ou le contenu d'un message.

Pairs restaurés. L'adversaire peut perturber un pair honnête. Il peut le soumettre à des lenteurs arbitraires. Il ne peut toutefois pas empêcher un pair de progresser dans l'exécution du protocole. L'adversaire peut également produire des pannes qui rendent le pair momentanément indisponible. Nous supposons que les pairs honnêtes disposent d'une mémoire durable et fiable de sorte à ce qu'ils retrouvent l'état dans lequel ils se trouvaient juste avant leur dernière panne. Pour les mêmes raisons que celles évoquées dans le profil *réseau asynchrone non-fiable*, l'adversaire ne peut pas provoquer des pannes dans l'objectif d'arrêter l'exécution définitive du protocole ou d'empêcher indéfiniment les pairs honnêtes de s'échanger un nombre non-borné de messages.

Pairs malintentionnés. L'adversaire contrôle un ensemble de pairs malintentionnés [6] qui peuvent librement travailler de concert. Il peut librement connecter de nouveaux pairs malintentionnés et déconnecter, éventuellement indéfiniment, des pairs malintentionnés. Les pairs malintentionnés peuvent se comporter de manière arbitraire.

L'adversaire peut utiliser l'ensemble de ses capacités et de ses ressources pour exprimer des comportements complexes. Toutefois, nous supposons les limitations suivantes :

— L'adversaire a des ressources limitées en puissance de calcul et en temps. Ces ressources ne sont pas suffisantes pour compromettre les primitives de cryptographies, telles que les signatures digitales et les fonctions de hachage résistantes aux collisions.

— Il ne peut pas empêcher un pair honnête d'authentifier la clef publique d'un autre pair honnête. L'adversaire ne peut pas obtenir la clef privée d'un pair honnête et il ne peut donc pas usurper son identité cryptographique.

Les protocoles de réplication optimiste s'intéressent généralement à un adversaire qui a les profils *réseau asynchrone non-fiable* et *pairs restaurés*. En ajoutant les deux autres profils *réseau corrompu* et *pairs malintentionnés* nous obtenons un adversaire qui est Byzantin [50].

6. Dans la littérature, le terme « défectueux » est également utilisé.

Certain·e·s auteur·ice·s font la distinction entre les pairs malintentionnés et les pairs honnêtes-mais-défaillants. Un pair honnête-mais-défaillant expose des comportements qui peuvent être prêtés à un pair malintentionné. Un pair honnête n'a pas la capacité de décider si un comportement nuisible d'un pair est le résultat d'une défaillance ou de motivations malintentionnées. Nous considérons donc les pairs honnêtes-mais-défaillants comme des pairs malintentionnés. Nous supposons qu'il existe au moins un pair honnête au sein de la collaboration.

2.5 Synthèse et objectifs

Un protocole de réplication optimiste autorise la modification et l'interrogation des copies d'un contenu répliqué sans aucune coordination. Les copies sont donc toujours disponibles pour être modifiées et interrogées. La modification concurrente des copies conduit inévitablement à leur divergence. La convergence des copies est une propriété essentielle des infrastructures de collaboration. Elle conditionne le succès d'une collaboration. Afin d'assurer des latences faibles et un passage à l'échelle du protocole, nous nous intéressons à des propriétés de convergence qui ne nécessitent pas de coordination.

Le passage à l'échelle et les latences faibles des protocoles de réplication optimiste auxquels nous nous intéressons permettent le support de collaboration massive. Ce type de collaboration a une probabilité plus importante de rencontrer des comportements malintentionnés. Nous avons choisi de nous intéresser à des protocoles qui résistent à un adversaire Byzantin. Ce dernier contrôle le réseau et un ensemble de pairs malintentionnés. Il peut également perturber le fonctionnement des pairs honnêtes.

Dans ce manuscrit nous cherchons à assurer la convergence des copies des pairs honnêtes sans la nécessité de coordination.

Chapitre 3

Fondations

Sommaire

Dans ce chapitre, nous introduisons les formalismes et les concepts que nous utilisons pour définir les contributions du chapitre 4 et du chapitre 5.

Les infrastructures pair-à-pair de collaboration permettent à un nombre important d'individus de collaborer. Pour ce faire, elles reposent sur des protocoles de réplication optimiste. Les protocoles de réplication optimiste garantissent des *propriétés de cohérence* des copies aux pairs et par extension aux collaborateur·ice·s. La convergence des copies est l'une de ces propriétés de cohérence. La section 3.1 introduit le formalisme que nous utilisons pour décrire les propriétés de cohérence. Notre formalisme étend le formalisme de Burckhardt [51] pour prendre en compte la présence d'un adversaire actif. Les protocoles de réplication garantissent souvent des ensembles communs de propriétés de cohérence. Pour faciliter l'appréciation de leurs similitudes, les propriétés de cohérence sont réunies sous des ensembles bien définis nommés *modèle de cohérence*. La section 3.1 présente un moyen de comparer les modèles de cohérence. La section 3.2 présente le modèle de cohérence causale ainsi que les modèles de cohérence sur lesquelles nous nous reposons pour établir la contribution du chapitre 4.

La convergence des copies est une propriété essentielle des protocoles de réplication optimiste. Dans le chapitre 2, nous avons motivé notre intérêt pour des protocoles qui permettent la convergence des copies sans coordination. De tels protocoles n'imposent pas aux pairs de communiquer pour s'entendre sur la manière de résoudre les conflits de modification qui surviennent au cours de la collaboration. Ils n'ont d'autres choix que d'adopter des résolutions prédéterminées de conflits. Les protocoles de réplication optimiste décident en général la manière de résoudre un conflit. Pour faciliter la comparaison des protocoles et leur implémentation, les stratégies de résolution de conflits sont encapsulées au sein de types de données répliquées nommés Conflict-free Replicated Data Types (CRDTs). Les concepteur·ice·s d'applications peuvent ainsi utiliser des types de données prêts à l'emploi. La section 3.3 présente ces types de données et illustre plusieurs sémantiques de résolution de conflits.

3.1 Spécification de modèles de cohérence

Un protocole détermine comment un pair peut interagir de manière conforme avec le système. En délimitant le périmètre d'action des pairs, le protocole garantit des propriétés de cohérence des copies. Burckhardt [51] a introduit un formalisme pour décrire des propriétés de cohérence et modéliser l'exécution d'un protocole. Le formalisme qu'il introduit n'est pas adapté aux environnements soumis à la présence de pairs malintentionnés. Dans la sous-section 3.1.1 nous présentons son formalisme avec quelques simplifications. Nous l'étendons dans la sous-section 3.1.2 pour permettre la description d'exécutions soumises à la présence de comportements malintentionnés. Dans le chapitre 2 nous avons distingué les propriétés de sûreté des propriétés de vivacité. Dans la sous-section 3.1.3 nous montrons comment reconnaître et spécifier ces deux genres de propriétés au sein du formalisme introduit.

3.1.1 Spécification en l'absence de pairs malintentionnés

Les collaborateur·ice·s s'attendent à ce que les infrastructures de collaboration leur offrent un certain nombre de garanties de cohérence des copies du contenu partagé. La figure 3.1 illustre un scénario dans lequel Bea s'attend à ce que Alice lise les messages textuels dans l'ordre dans lequel elle les a écrits. Certains protocoles de réplication optimiste peuvent respecter les attentes de Bea et d'autres peuvent ne pas les respecter. Les attentes des collaborateur·ice·s sont exprimées sous la forme de propriétés de cohérence.

Les propriétés de cohérence sont de différentes natures. Certaines contraignent l'ordre dans lequel les modifications sont intégrées par les pairs. D'autres spécifient la valeur retournée par les interrogations des pairs. Elles expriment toutes des garanties qui sont offertes aux collaborateur·ice·s. De ce fait, elles s'expriment sur les interactions des collaborateur·ice·s avec le protocole et sur les réponses que le protocole fournit aux collaborateur·ice·s. Les collaborateur·ice·s interagissent avec le système en exécutant des opérations. L'ensemble des opérations exécutées forme une *histoire*. L'histoire de la figure 3.1 est constituée des trois opérations de Alice (pair p_A) et des trois opérations de Bea (pair p_B).

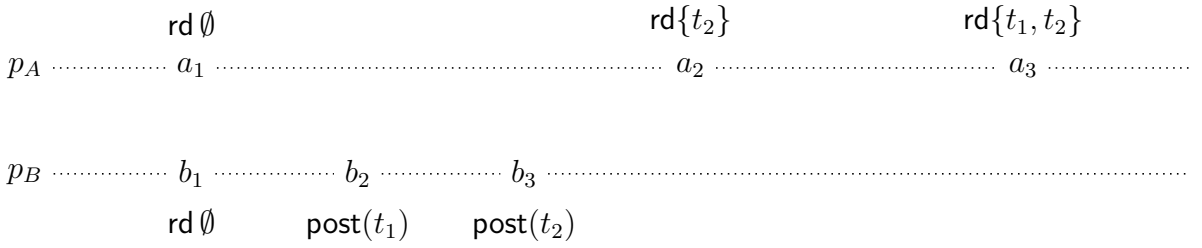


FIGURE 3.1 – Alice (pair p_A) et Bea (pair p_B) discutent sur un fil de discussion répliqué. Le fil de discussion répliqué supporte une opération d'ajout d'un message textuel $\{\mathsf{post}(\mathsf{txt}) \mid \mathsf{txt} \in \mathsf{String}\}$, et une opération de lecture rd. L'exécution d'une opération de lecture retourne l'ensemble des messages qui ont été ajoutés. Ainsi l'exécution de l'opération $\mathsf{post}(t_1)$ permet d'ajouter le message textuel t_1 sur le fil de discussion. Nous notons $\mathsf{rd}\, s$ l'exécution d'une opération de lecture suivie de l'ensemble s de messages textuels qu'elle retourne. Bea ajoute un premier message t_1 suivi d'un second message t_2. Elle s'attend à ce que Alice prenne connaissance du premier message avant de prendre connaissance du second message. Cependant, le protocole ne lui garantit pas cette propriété. Alice lit le second message avant de lire le premier.

Les collaborateur·ice·s exécutent des opérations de modification pour altérer l'état du contenu partagé et exécutent des opérations d'interrogation pour observer les effets de leurs modifications. Nous supposons que tout type de contenu partagé est interrogé avec une unique opération rd. Il s'agit de l'*opération de lecture*.

Nous supposons que l'exécution d'une opération se termine et qu'elle se déroule sur un intervalle de temps non-nul. Les dates de début et de fin d'exécution des opérations sont issues d'une unique horloge à temps continu. Cette horloge n'a pas d'existence propre au cours d'une exécution réelle. Nous avons toutefois besoin d'un temps continu et universel pour exprimer certaines propriétés de cohérence.

Lorsque nous raisonnons sur l'exécution d'opérations, nous nous intéressons souvent à l'ordre dans lequel les opérations sont exécutées. Les dates de début et de fin d'exécution d'une opération fournissent plus d'information que nécessaire. Pour faciliter le raisonnement sur l'ordre d'exécution des opérations d'une histoire, nous introduisons la relation suivante :

Retourne-avant [51]. Une relation d'ordre partiel qui rend compte des opérations dont l'exécution ne coïncide pas dans le temps. Une opération x retourne-avant une opération y, et nous écrivons $x \xrightarrow{\mathsf{rb}} y$, si et seulement si l'exécution de x se termine avant que l'exécution de y débute. Si x ne retourne pas avant y et si y ne retourne pas avant x, alors leur exécution coïncide. La figure 3.2 montre un exemple de deux opérations dont l'exécution coïncide dans le temps.

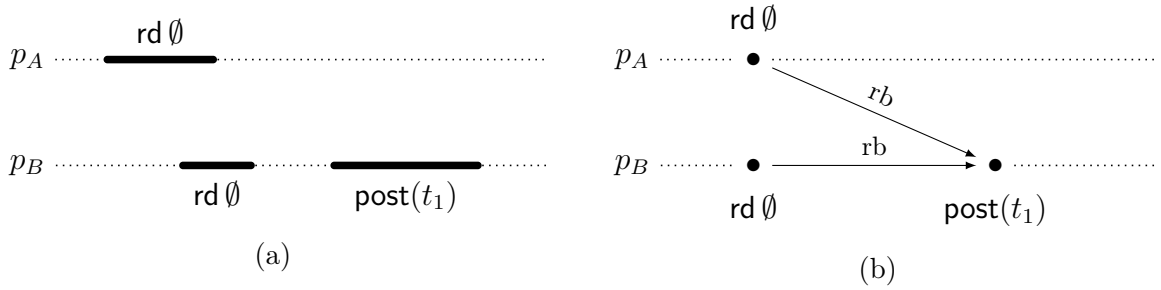


FIGURE 3.2 – (a) L'intervalle de temps sur lequel l'opération d'interrogation de Alice (p_A) s'exécute a une intersection non-nulle avec l'intervalle de temps sur lequel l'opération d'interrogation de Bea (p_B) s'exécute. Ces deux opérations coïncident donc dans le temps. En revanche, l'intervalle de temps sur lequel l'opération de modification de Bea s'exécute a une intersection nulle avec les deux opérations d'interrogation. Elle ne coïncide donc pas dans le temps avec ces deux opérations d'interrogation. Dans ce manuscrit, les propriétés de cohérence que nous exprimons ne nécessitent pas la connaissance de l'intervalle de temps sur lequel une opération s'exécute. (b) Nous contractons donc les intervalles en des points. Deux points alignés verticalement correspondent à des opérations dont l'exécution coïncide dans le temps. Si l'exécution de deux opérations ne coïncide pas, alors une opération *retourne-avant* l'autre. Ici, cette information est explicitement marquée par l'utilisation de la relation *retourne-avant* $\xrightarrow{\mathsf{rb}}$.

La Definition 3.1 définit formellement une histoire. Elle définit en particulier les caractéristiques associées aux opérations. Le tableau 3.1 donne les caractéristiques des opérations de l'histoire de la figure 3.1.

Definition 3.1 (Histoire)**.** [51] Une *histoire* d'un contenu partagé de type T est un n-uplet $H \stackrel{\text{def}}{=} \langle E, \mathsf{peer}, \mathsf{call}, \mathsf{rval}, \mathsf{rb}\rangle$ tel que E est un ensemble dénombrable d'opérations. Pour toute opération x :

— $\mathsf{peer}(x) \in \mathsf{Peers}$ est l'auteur de l'opération x.

— $\mathsf{call}(x) \in \mathsf{Op}_T$ est l'appel effectué par l'auteur de x. Ces appels dépendent de T.

— $\mathsf{rval}(x) \in \mathsf{Val}_T$ est la valeur de retour de l'appel de x s'il s'agit d'une interrogation. Ces valeurs de retour dépendent de T.

$\mathsf{rb} \subseteq E \times E$ est la relation *retourne-avant*. Il s'agit d'un ordre partiel localement fini [7] qui peut être interprété sur une ligne de temps [8] :

$$u \xrightarrow{\mathsf{rb}} v \wedge y \xrightarrow{\mathsf{rb}} z \implies u \xrightarrow{\mathsf{rb}} z \vee y \xrightarrow{\mathsf{rb}} v$$

Un protocole garantit une propriété de cohérence si et seulement si toutes les histoires qu'il peut produire vérifient cette propriété. Une histoire vérifie une propriété de cohérence si et seulement si ses caractéristiques (valeur de retour des opérations, ordre d'exécution, ...) peuvent être expliquées par l'ajout de la relation suivante :

7. Pour tout couple d'opérations, l'ensemble des opérations comprises entre ces 2 opérations est fini.
8. GREENOUGH [52] parle d'ordre d'intervalle.

opération	peer	call	rval
a_1	p_A	rd	$\emptyset$
a_2	p_A	rd	$\{t_2\}$
a_3	p_A	rd	$\{t_1, t_2\}$
b_1	p_B	rd	$\emptyset$
b_2	p_B	post(t_1)	
b_3	p_B	post(t_2)	

TABLE 3.1 – Caractéristiques des opérations exécutées dans l'histoire de la figure 3.1. Un fil de discussion supporte une opération de lecture et une opération de modification $\mathsf{Op}_T = \{\mathsf{rd}\} \cup \{\mathsf{post}(\mathsf{msg}) \mid \mathsf{msg} \in \mathsf{String}\}$. Les valeurs de retour correspondent à des ensembles de messages $\mathsf{Val}_T = \mathcal{P}_{\mathsf{fin}}(\mathsf{String})$. $\mathcal{P}_{\mathsf{fin}}(\mathsf{String})$ est l'ensemble des sous-ensembles finis des chaînes de caractères.

Visibilité [51]. Une relation qui rend compte de l'intégration des effets des opérations de modification par les pairs. Si une opération de modification x est visible à une opération d'interrogation y, alors x a (éventuellement) un effet sur la valeur de retour de y. Nous disons aussi que l'effet de x a été intégré à la copie avant l'exécution de l'opération y sur cette même copie. Dans ce cas, nous écrivons $x \xrightarrow{\mathsf{vis}} y$. La relation de visibilité n'est pas forcément transitive. Par exemple, dans la figure 3.3, b_2 est visible à b_3 et b_3 est visible à a_2, mais b_2 n'est pas visible à a_2. Par souci de concision, nous écrivons $x \leq_{\mathsf{vis}} y$ si x est visible ou est égal à y.

Une histoire augmentée de la relation de *visibilité* consiste en une *exécution abstraite* [51]. La Definition 3.2 définit formellement une exécution abstraite. La figure 3.3 est une exécution abstraite de l'histoire de la figure 3.1.

Definition 3.2 (Exécution abstraite)**.** [51] Une *exécution abstraite* d'un contenu partagé de type T est un n-uplet $A \stackrel{\text{def}}{=} \langle H, \mathsf{vis} \rangle$ tel que :

— H est une histoire d'un contenu partagé de type T.

— $\mathsf{vis} \subseteq H \times H$ est la relation de *visibilité*. Elle est sans cycle et localement finie.

Une histoire vérifie une propriété de cohérence si et seulement si il existe au moins une exécution abstraite qui la respecte. Il suffit donc de générer toutes les exécutions abstraites possibles pour une histoire donnée et vérifier qu'au moins l'une d'entre elles respecte la propriété de cohérences considérée. La figure 3.4 met en évidence les relations entre histoires, exécutions abstraites, et propriétés de cohérence.

La relation de visibilité rend compte de l'intégration des effets des opérations de modification par les pairs. En tant que tel, seules les relations de visibilité dirigées d'une opération de modification vers une opération d'interrogation semblent faire sens. Dans la figure 3.3 on remarque ainsi que l'effet de l'opération b_3 est pris en compte par l'opération de lecture a_2, et les effets des opérations b_2 et b_3 sont pris en compte par l'opération de lecture a_3. Notre définition de la relation de visibilité nous permet de mettre en relation tout type d'opérations. Dans la figure 3.3 il existe ainsi une relation de visibilité entre les

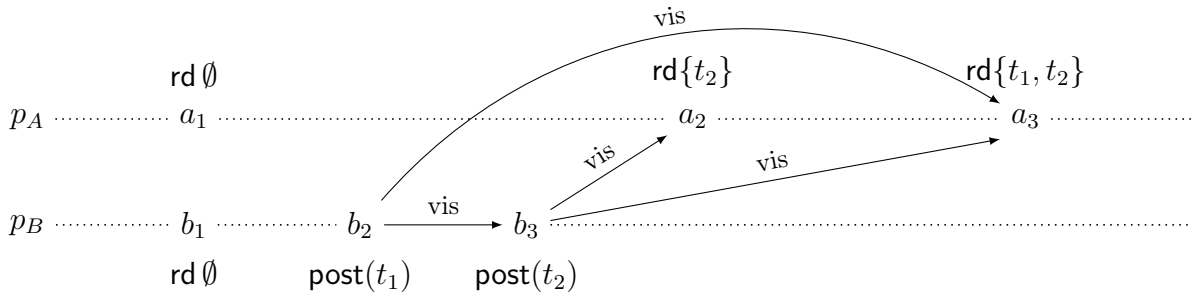


FIGURE 3.3 – Une *exécution abstraite* de l'histoire de la figure 3.1. La relation *retourne-avant* est inférée par le positionnement horizontal des opérations.

opérations de modification b_2 et b_3. Cette flexibilité nous permet de simplifier l'expression des propriétés de cohérence.

Les relations précédemment définies sont suffisantes pour exprimer toutes les propriétés de cohérence auxquelles nous nous intéressons dans ce manuscrit [9]. Ces expressions peuvent parfois être peu lisibles. Par exemple, pour spécifier le type de retour d'une opération de lecture, il est nécessaire de prendre en compte l'ensemble des opérations de modification qui sont visibles à cette opération de lecture. Dans la figure 3.3, si un ajout de message textuel est visible à une opération de lecture, alors cette dernière inclut ce message dans sa valeur de retour. Pour faciliter l'expression de telles propriétés, nous introduisons dans la Definition 3.3 le *contexte* d'une opération. Le contexte d'une opération regroupe les opérations de modification qui lui sont visibles. Le tableau 3.2 donne le contexte des opérations de l'exécution abstraite de la figure 3.3.

Definition 3.3 (Contexte)**.** Le *contexte* d'une opération x dans une *exécution abstraite* A, dénoté par $\mathsf{ctx}_A(x)$, correspond à l'ensemble des opérations de modification [10] qui lui sont visibles.

$$\mathsf{ctx}_A(x) \stackrel{\text{def}}{=} \left\{ y \in A \;\middle|\; y \xrightarrow{\mathsf{vis}} x \wedge \mathsf{call}(y) \neq \mathsf{rd} \right\}$$

La spécification du type de données répliquées de la figure 3.1 (fil de discussion) peut ainsi être exprimée par l'équation 3.1 :

$$x \in A \wedge \mathsf{call}(x) = \mathsf{rd} \implies \mathsf{rval}(x) = \{t \mid \exists y \in \mathsf{ctx}_A(x).\, \mathsf{call}(y) = \mathsf{post}(t)\} \tag{3.1}$$

L'exécution abstraite de la figure 3.3 respecte cette propriété de cohérence. L'histoire de la figure 3.1 respecte donc cette propriété de cohérence.

Le formalisme présenté nous permet de définir des propriétés de cohérence comme de simples prédicats sur des exécutions abstraites. Des propriétés de cohérence sont garanties

9. BURCKHARDT [51] ajoute également une relation d'arriération ar à ses exécutions abstraites. Elle s'avère utile dans la définition de certains type de données répliquées. Nous l'avons délibérément omise par souci de simplification.

10. BURCKHARDT [51] et VIOTTI et al. [53] incluent également les opérations d'interrogation dans le contexte d'une opération. Nous les excluons pour simplifier l'expression de certaines propriétés de cohérence.

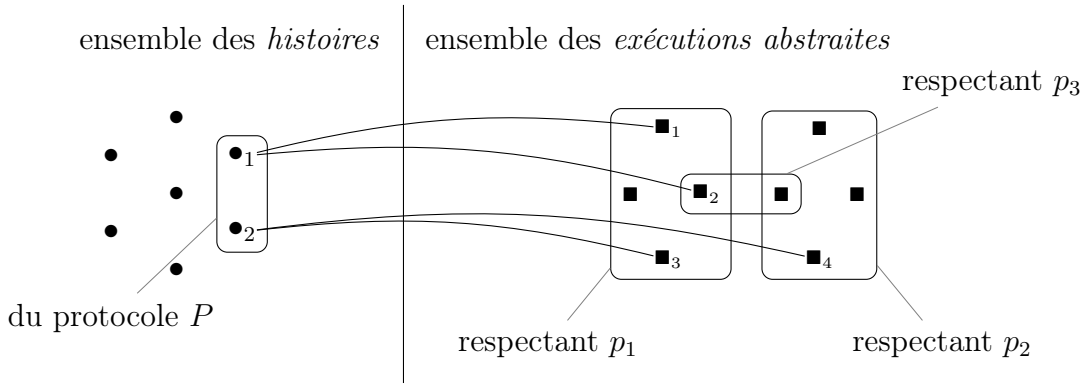


FIGURE 3.4 – Illustration des relations entre protocoles, histoires, exécutions abstraites, et propriétés de cohérence. L'exécution du protocole P peut produire les histoires $\bullet_1$ et $\bullet_2$. $\blacksquare_1$ et $\blacksquare_2$ correspondaient aux exécutions abstraites que nous obtenons à partir de $\bullet_1$. $\blacksquare_3$ et $\blacksquare_4$ correspondaient aux exécutions abstraites que nous obtenons à partir de $\bullet_2$. L'histoire $\bullet_1$ vérifie les propriétés de cohérence p_1 et p_3 puisque l'exécution abstraite $\blacksquare_2$ les respecte. De même, l'histoire $\bullet_2$ vérifie la propriété p_1 ou p_2 puisque l'exécution abstraite $\blacksquare_3$ respecte p_1 et l'exécution abstraite $\blacksquare_4$ respecte p_2. Le protocole P garantit ainsi la propriété p_1 puisque chacune de ses histoires est associée à au moins une exécution abstraite qui la respecte.

Opération	ctx	obs
a_1	$\emptyset$	$\{p_A\}$
a_2	$\{b_3\}$	$\{p_A\}$
a_3	$\{b_2, b_3\}$	$\{p_A\}$
b_1	$\emptyset$	$\{p_B\}$
b_2	$\emptyset$	$\{p_A, p_B\}$
b_3	$\emptyset$	$\{p_A, p_B\}$

TABLE 3.2 – Contexte et observateurs des opérations de l'exécution abstraite de la figure 3.3.

par un protocole si chaque histoire du protocole peut être augmentée en une exécution abstraite qui les respecte toutes. BURCKHARDT [51] a défini les histoires et les exécutions abstraites avec l'hypothèse que les pairs suivent le protocole. Nous proposons de généraliser le formalisme pour qu'il puisse être utilisé en présence de pairs qui peuvent déroger au protocole. Nous qualifions ces pairs de malintentionnés.

3.1.2 Spécification en présence de pairs malintentionnés

Dans cette section nous proposons une extension du formalisme de BURCKHARDT [51]. Cette extension permet de décrire des exécutions dans lesquels des pairs peuvent déroger au protocole. Nous qualifions ces pairs de *malintentionnés*. Inversement, les pairs qui suivent le protocole sont qualifiés d'*honnêtes*. Nous notons Honest et Malicious l'ensemble des pairs honnêtes et malintentionnés. L'union de ces deux ensembles correspond à l'ensemble des

pairs Peers. Ces ensembles sont connus, étant donné que nous décrivons des exécutions d'un point de vue global. L'ensemble des pairs contient au moins un pair honnête ($\mathsf{Honest} \neq \emptyset$).

Sans perte de généralité, nous supposons que les pairs honnêtes exécutent les opérations de manière séquentielle. Les opérations exécutées sur un même pair sont donc linéairement ordonnées selon la relation *retourne-avant*. Elles forment une énumération. Les opérations d'un pair honnête peuvent donc être numérotées dans l'ordre dans lequel elles sont exécutées. Dans la figure 3.3 on observe ainsi deux énumérations $a_1 \xrightarrow{\mathsf{rb}} a_2 \xrightarrow{\mathsf{rb}} a_3$ et $b_1 \xrightarrow{\mathsf{rb}} b_2 \xrightarrow{\mathsf{rb}} b_3$. Une histoire qui remplie cette condition est une *histoire bien-formée*. La Definition 3.4 définit formellement une histoire bien-formée.

Definition 3.4 (Histoire bien-formée)**.** En présence de pairs malintentionnés, une histoire H est *bien-formée* si et seulement si pour tout pair honnête h, la projection de la relation *retourne-avant* sur h forme une énumération.

$$x, y \in H \wedge \mathsf{peer}(x) \in \mathsf{Honest} \wedge \mathsf{peer}(x) = \mathsf{peer}(y) \implies x \xrightarrow{\mathsf{rb}} y \vee y \xrightarrow{\mathsf{rb}} x$$

Les pairs malintentionnés peuvent librement déroger à un protocole. Ils peuvent par exemple :

— Exécuter des opérations qui ne se terminent pas

— Exécuter des opérations qui ne sont pas reconnues par le protocole

— Produire des interactions qui ne peuvent pas être caractérisées par des opérations

Un protocole offre des garanties de cohérence uniquement aux pairs qui le suivent, c'est-à-dire aux pairs honnêtes. Ces garanties se vérifient donc sur ce que les pairs honnêtes prennent en compte. Seuls les interactions acceptées par les pairs honnêtes sont incluses dans une histoire.

Les pairs honnêtes acceptent uniquement les interactions reconnues par le protocole. En d'autres termes ils rejettent toutes les interactions qui ne correspondent pas à des opérations définies par le protocole. Au sein d'une exécution abstraite, une opération est acceptée par les pairs honnêtes si et seulement si elle est exécutée par un pair honnête ou si elle est visible à au moins une opération exécutée par un pair honnête. Une opération acceptée est ainsi observée par au moins un pair honnête. La Definition 3.5 définit ce qu'est un observateur d'une opération. Le tableau 3.2 donne les observateurs de chaque opération de l'histoire de la figure 3.1.

Definition 3.5 (Observateurs)**.** Soit une opération x d'une exécution abstraite A. Un pair p est un observateur de x si et seulement si (i) p a exécuté x ou (ii) p a exécuté une opération de A telle que x est visible à cette dernière. $\mathsf{obs}_A(x)$ désigne l'ensemble des observateurs de x.

$$\mathsf{obs}_A(x) \stackrel{\text{def}}{=} \{\mathsf{peer}(y) \mid y \in A \wedge x \leq_{\mathsf{vis}} y\}$$

Une *exécution abstraite bien-formée* inclut uniquement les opérations qui sont observées par les pairs honnêtes. Ces opérations correspondent (i) aux opérations exécutées par les pairs honnêtes et (ii) aux opérations exécutées par des pairs malintentionnés qui sont visibles à au moins une opération d'un pair honnête. La Definition 3.6 définit formellement une exécution abstraite bien-formée. La figure 3.5 donne un exemple d'exécution abstraite mal-formée et d'une exécution abstraite bien-formée.

Definition 3.6 (Exécution abstraite bien-formée)**.** En présence de pairs malintentionnés, une *exécution abstraite* $A \stackrel{\text{def}}{=} \langle H, \mathsf{vis} \rangle$ est *bien-formée* si et seulement si :

BF1 H est une *histoire bien-formée.*

BF2 Les opérations dans E des pairs malintentionnés sont chacune visible à au moins une opération d'un pair honnête.

$$x \in A \wedge \mathsf{peer}(x) \notin \mathsf{Honest} \implies \mathsf{obs}_A(x) \cap \mathsf{Honest} \neq \emptyset$$

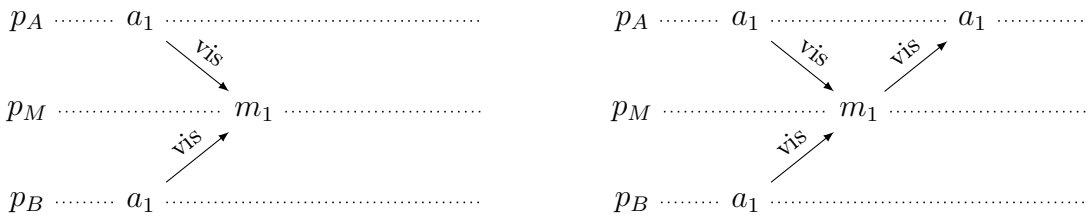


FIGURE 3.5 – Exemples d'une exécutions abstraite mal-formée et d'une exécution abstraite bien-formée. Soit un groupe de pairs composé d'un pair malintentionné p_M et de deux pairs honnêtes p_A et p_B. L'exécution abstraite (a) n'est pas bien-formée. L'opération m_1 est uniquement observée par un pair malintentionné p_M. L'exécution abstraite (b) est bien-formée. L'opération m_1 est observée par le pair honnête p_A.

Remarque. Dans la suite de ce manuscrit nous considérons uniquement, sauf explicitement indiqué, des histoires et des exécutions abstraites bien-formées. Par souci de concision, nous omettons de préciser qu'une histoire ou une exécution abstraite est bien-formée.

Les pairs malintentionnés peuvent faire coïncider dans le temps l'exécution de plusieurs opérations. La figure 3.6 illustre ce scénario. Pour pouvoir numéroter les opérations des pairs malintentionnés, nous utiliserons des exemples dans lesquels les pairs malintentionnés exécutent leurs opérations de manière séquentielle.

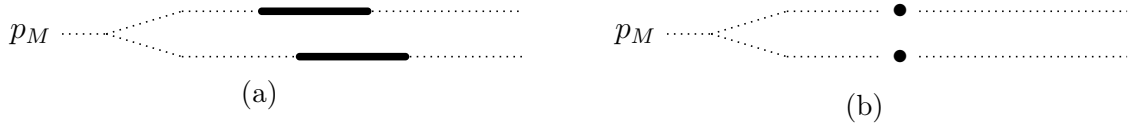


FIGURE 3.6 – Mallory est une collaboratrice malintentionnée. Elle est attachée au pair p_M. (a) Les exécutions des deux opérations de Mallory coïncident dans le temps. (b) Exemple d'une exécution abstraite correspondante.

La généralisation du formalisme de BURCKHARDT [51] que nous proposons nous permet de décrire des exécutions qui incluent des pairs malintentionnés. Ces descriptions nous permettent de rendre compte des propriétés de cohérence qu'un protocole garantit en présence de pairs malintentionnés.

3.1.3 Vivacité et sûreté

La littérature divise les propriétés de cohérence entre propriétés de sûreté et propriétés de vivacité. Une propriété de sûreté décrit « quelque chose de mauvais » qui ne devrait jamais survenir [54]. Elle doit être respectée en tout point de l'exécution d'un protocole. Il s'agit d'un invariant. Une propriété de vivacité décrit « quelque chose de positif » qui devrait survenir [54]. Elle peut être violée en plusieurs points de l'exécution d'un protocole. Une propriété de cohérence peut être la conjonction d'une propriété de sûreté et d'une propriété de vivacité [45].

Dans ce chapitre nous avons déjà rencontré plusieurs propriétés de sûreté telles que l'équation 3.1 qui spécifie le fil de discussion. En revanche, nous n'avons pas rencontré de propriétés de vivacité. Dans le chapitre 2, l'intégration à terme des modifications est une propriété de vivacité. Elle stipule qu'une modification est à terme intégrée par tous les pairs.

Parce que les propriétés de vivacité décrivent quelque chose qui devrait survenir, elles nous invitent à penser l'exécution d'un protocole comme une succession infinie d'événements qui se déroule dans le temps. Une histoire peut décrire les opérations exécutées sur l'intégralité ou une partie d'une exécution. Elle décrit une partie d'une exécution lorsqu'elle enregistre l'ensemble des opérations exécutées depuis le commencement de l'exécution jusqu'à un temps donné. Une histoire peut évoluer vers plusieurs histoires hypothétiques dans lesquelles de nouvelles opérations ont été exécutées. Par abus de langage nous parlons d'*opérations futures*. Ces histoires hypothétiques sont des *extensions de l'histoire*. La Definition 3.7 définit formellement l'extension d'une histoire.

Definition 3.7 (Extension d'une histoire)**.** Une histoire H' est une extension d'une histoire H, et nous écrivons $H' \sqsupset H$, si et seulement si :

— Les opérations de H correspondent à un sous-ensemble fini des opérations de H'. H est donc une histoire finie.

$$H \subset H' \wedge H \textit{ est finie}$$

— Si l'opération y est incluse dans H, alors toutes les opérations de H' qui retournent-avant y sont également incluses dans H.

$$y \in H \wedge x \xrightarrow{\mathsf{rb}}_{H'} y \implies x \in H$$

— Les relations *retourne-avant* sont inchangées.

$$x, y \in H \wedge x \xrightarrow{\mathsf{rb}}_{H'} y \iff x \xrightarrow{\mathsf{rb}}_{H} y$$

— Les caractéristiques des opérations (pair, appel, valeur de retour) sont conservées.

Une propriété de vivacité peut alors être exprimée comme un prédicat qui doit être vérifié dans l'histoire considérée ou dans une ou plusieurs de ses extensions. Une propriété de vivacité ne peut pas être vérifiée indépendamment des autres propriétés de cohérence considérées. Seules les extensions de l'histoire qui respectent les autres propriétés de cohérence doivent être considérées.

L'intégration à terme des modifications pourrait ainsi être traduite par : une opération de modification est à terme visible à une infinité d'opérations d'interrogation. Nous pouvons tirer avantage de la flexibilité de la relation de visibilité pour simplifier l'expression à : une opération est à terme visible à une infinité d'opérations. En d'autres termes, pour toute opération il n'existe pas une infinité d'opérations auxquelles elle n'est pas visible :

$$\forall x \in A. \left\{ y \in A \;\middle|\; x \not\xrightarrow{\mathsf{vis}} y \right\} \textit{est fini} \tag{3.2}$$

La classification des propriétés de cohérence en propriétés de vivacité et en propriétés de sûreté permet de mieux apprécier les garanties qu'elles expriment.

3.1.4 Ordre des Modèles de cohérence

Pour apprécier la différence entre deux protocoles de réplication, il suffit d'énumérer l'ensemble des propriétés de cohérence que chacun garantit. La comparaison de ces ensembles de propriétés de cohérence peut être fastidieux. C'est pour cette raison que les propriétés de cohérence sont généralement réunies au sein de *modèles de cohérence*. Un protocole garantit un *modèle de cohérence* si toutes les histoires qu'il peut produire peuvent être justifiées par au moins une *exécution abstraite* qui respecte ce modèle de cohérence. La Definition 3.8 introduit un abus de langage qui permet de désigner l'ensemble des exécutions abstraites qui respectent un modèle de cohérence par le nom du modèle de cohérence.

Definition 3.8 (Modèle de cohérence et exécutions abstraites)**.** Soit un modèle de cohérence C. Nous écrivons $A \in C$ si et seulement si l'exécution abstraite A respecte le modèle de cohérence C. C désigne donc également l'ensemble des exécutions abstraites qui respectent le modèle de cohérence C.

Les modèles de cohérences offrent des garanties variées. Certains modèles de cohérence permettent des exécutions que d'autres ne permettent pas. Un modèle de cohérence est dit plus fort qu'un second modèle de cohérence s'il accepte un sous-ensemble strict des exécutions abstraites que le premier accepte. Dans la Definition 3.9, nous introduisons la notation que nous utilisons pour mettre en évidence qu'un modèle de cohérence est plus fort qu'un autre. Deux modèles de cohérence sont incompatibles si l'un n'est pas plus fort que l'autre. Nous pouvons néanmoins juger de la « force » d'un modèle de cohérence par rapport au nombre d'exécutions abstraites qu'il accepte. L'ensemble des modèles de cohérence est alors partiellement ordonné [53].

Definition 3.9 (Force des modèles de cohérence)**.** Soit deux modèle de cohérence C et C'. Le modèle de cohérence C est plus fort que le modèle de cohérence C', si et seulement si l'ensemble des exécutions abstraites qui respectent C est strictement inclus dans l'ensemble des exécutions abstraites qui respectent C'. En d'autres termes $C \subset C'$.

Un modèle de cohérence plus fort permet de limiter le nombre de comportements possibles. Par conséquent, il permet de raisonner plus facilement lors de la conception et l'utilisation d'un protocole. La recherche des modèles les plus forts pour un modèle de système donné est un domaine de recherche actif [47, 48, 53]. Dans la section 3.2 nous présentons des modèles de cohérence.

3.2 Modèles de cohérence et causalité

Dans cette section nous présentons plusieurs modèles de cohérence. Nous décrivons d'abord le modèle de cohérence à terme forte qui a été abordé informellement dans le chapitre 2. Nous décrivons ensuite le modèle de cohérence causale. Les modèles de cohérence que nous présentons dans la suite de ce manuscrit ne reposent pas directement sur le modèle de cohérence causale. Cependant, il s'agit d'un classique de la littérature des systèmes distribués. En outre, sa description nous permet d'illustrer l'utilisation du formalisme présenté dans la section 3.1 et sa compréhension conditionne la compréhension des modèles de cohérence que nous abordons par la suite. Nous décrivons également le modèle de cohérence View-Fork-Join-Causal (VFJC) sur lequel nous nous reposons pour établir la contribution du chapitre 4. Pour faciliter son introduction, nous présentons le modèle de cohérence Fork-Join-Causal (FJC).

3.2.1 Cohérence à terme forte

Le modèle de cohérence à terme est le modèle le plus faible qu'un protocole de réplication doit respecter [12, 51]. Dans ce manuscrit nous considérons un modèle de cohérence avec une propriété de convergence plus forte. Il s'agit du modèle de cohérence à terme forte [28, 51]. Nous avons motivé ce choix dans le chapitre 2. La Definition 3.10 définit ce modèle. Elle reprend la propriété de visibilité à terme définie dans la sous-section 3.1.3.

Definition 3.10 (Cohérence à terme forte)**.** Soit une exécution abstraite A. A respecte le modèle de cohérence à terme forte si et seulement si :

EC1 (visibilité à terme) Les effets des opérations de modification sont à terme intégrés par les pairs (honnêtes).

$$\forall x \in A. \left\{ y \in A \;\middle|\; x \overset{\mathsf{vis}}{\not\longrightarrow} y \right\} \textit{ est fini}$$

EC2 (Convergence forte) Les copies des pairs (honnêtes) convergent.

$$\forall x, y \in A. \, \mathsf{call}(x) = \mathsf{call}(y) = \mathsf{rd} \wedge \mathsf{ctx}_A(x) = \mathsf{ctx}_A(y) \implies \mathsf{rval}(x) = \mathsf{rval}(y)$$

Dans ce manuscrit, la propriété de convergence forte est impliquée par les spécifications de types de données répliquées. C'est le cas dans l'équation 3.1 qui spécifie le fil de discussion. Deux opérations de lecture qui ont le même contexte ont nécessairement la même valeur de retour.

L'ensemble des modèles de cohérence que nous abordons dans les sous-sections suivantes sont compatibles avec le modèle de cohérence à terme forte. Un protocole peut assurer l'un ou plusieurs de ces modèles ainsi que le modèle de cohérence à terme forte. Mahajan et al. [47] a prouvé ces compatibilités.

3.2.2 Cohérence causale

Le modèle de cohérence causale capture les relations de causalités potentielles entre les opérations et garantit que les pairs honnêtes intègrent les effets des opérations en respectant leurs causalités. Si une opération x cause potentiellement une opération y, alors tous les pairs honnêtes intègrent l'effet de x avant l'effet de y. Les pairs honnêtes s'accordent sur l'ordre d'intégration des effets des opérations causalement liées. Les opérations qui ne sont pas causalement liées sont dites concurrentes. Les pairs intègrent les opérations concurrentes dans des ordres potentiellement distincts. Par exemple si x et y sont deux opérations concurrentes et causent toutes deux l'opération z, alors un premier pair peut intégrer l'effet de x, puis l'effet de y, puis l'effet de z, et un second pair peut intégrer l'effet de y, puis l'effet de x, puis l'effet de z. x et y doivent être intégrées avant z.

Lamport [55] a introduit la définition communément acceptée de la causalité potentielle dans le cadre d'un système de passage de messages. Hutto et al. [56] ont par la suite traduit cette notion en propriétés de cohérence. Une opération x cause potentiellement une opération y si et seulement si (i) x *retourne-avant* y et x a le même auteur que y, ou (ii) L'effet de x est intégré par l'auteur de y avant l'exécution de y (x est visible à y), ou récursivement (iii) il existe une opération x' telle que x cause potentiellement x' et x' cause potentiellement y. La causalité potentielle est transitive.

Dans une exécution abstraite qui respecte le modèle de cohérence causale, si une opération x cause potentiellement une opération y, alors l'effet de x est intégré avant l'exécution de y. Il s'ensuit que chaque relation de causalité potentielle donne lieu à une relation de visibilité. La relation de visibilité devient donc transitive. La transitivité de la relation de visibilité nous permet de définir la relation de *concurrence* et la relation de *visibilité immédiate*. Ces relations simplifient l'expression de propriétés de cohérence que nous aborderons ultérieurement.

Concurrence. Lorsque la relation de visibilité est transitive, nous disons que deux opérations x et y sont concurrentes, et nous écrivons $x \| y$, si x n'est pas visible à y et y n'est pas visible à x. Dans la figure 3.7b, a_1 et b_1 sont concurrentes, ainsi que a_1 et b_2.

$$x \| y \overset{\text{def}}{\iff} x \not\xrightarrow{\mathsf{vis}} y \wedge y \not\xrightarrow{\mathsf{vis}} x$$

Visibilité immédiate [57, 58]. Lorsque la relation de visibilité est transitive, une opération x, qui est visible à une opération y, est immédiatement visible à y, et nous écrivons $x \downarrow y$, si et seulement si il n'existe pas une opération z telle que x est visible à z et z st visible à y. La relation de visibilité immédiate correspond à la réduction transitive de la relation de visibilité.

$$x \downarrow y \overset{\text{def}}{\iff} x \xrightarrow{\mathsf{vis}} y \wedge \nexists z.\, x \xrightarrow{\mathsf{vis}} z \xrightarrow{\mathsf{vis}} y$$

La figure 3.7a montre que la représentation graphique d'une exécution abstraite où la relation de visibilité est transitive peut devenir difficile à appréhender. Lorsque la relation de visibilité est transitive, nous préférons représenter la relation de visibilité immédiate. La figure 3.7a et la figure 3.7b sont ainsi équivalentes.

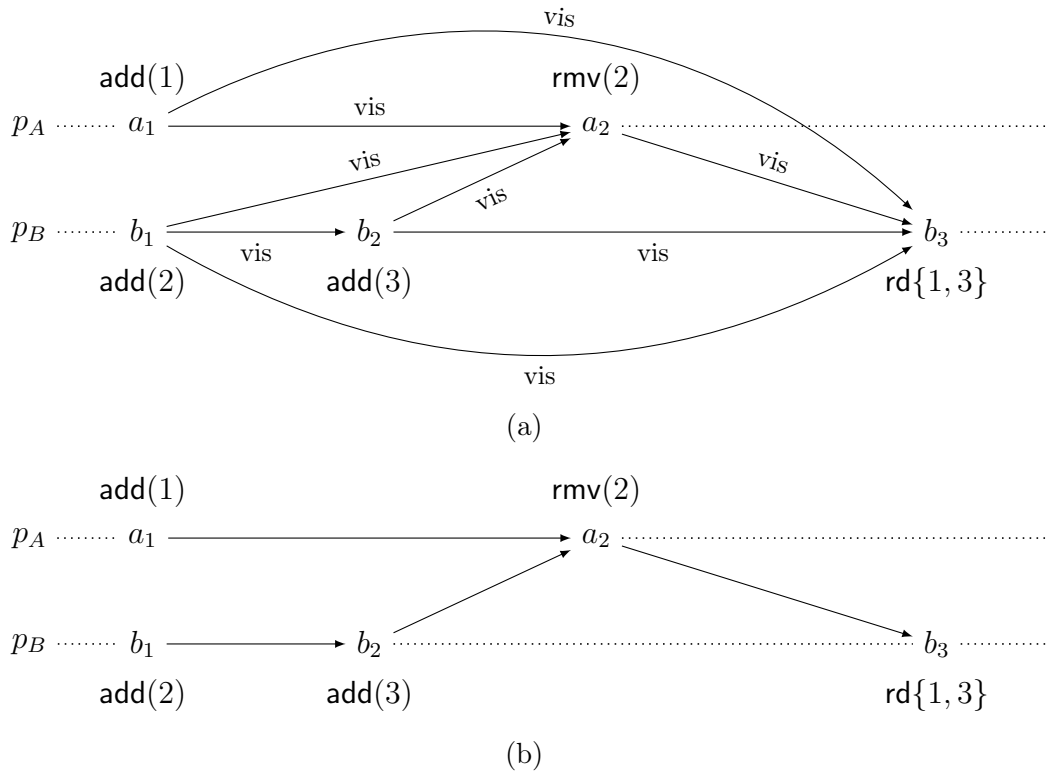


FIGURE 3.7 – Exécutions abstraites qui respectent le modèle de cohérence causale. L'exécution abstraite de la (b) correspond à la réduction transitive de l'exécution abstraite de la (a).

La littérature compte plusieurs spécifications de la cohérence causale. Certaines spécifications [59] imposent la manière de résoudre les conflits de modification. Similairement à MAHAJAN et al. [47], nous pensons qu'il est préférable de séparer les garanties d'ordre entre les opérations et les garanties concernant la résolution de conflits. Certaines spécifications [51, 53] procèdent à cette séparation, mais ajoutent une garantie qui restreint les résultats possibles des opérations de lecture. Notre Definition 3.11 de la cohérence causale se restreint uniquement à des garanties d'ordre.

Notre définition exclut les visibilités spéculatives. En d'autres termes, un pair ne peut pas observer une opération qui n'a pas encore été exécutée. Il ne peut donc pas spéculer sur les opérations qui seront éventuellement exécutées dans le futur. Cette garantie est nécessaire pour construire certaines propriétés. Nous parlons ainsi de cohérence causale non-spéculative [11] [47, 53]. En dehors de cette section, par abus de langage, nous parlerons de cohérence causale.

11. MAHAJAN et al. [47] parlent de *cohérence causale naturelle*. VIOTTI et al. [53] parlent de cohérence causale temps réel.

Definition 3.11 (Cohérence causale non-spéculative)**.** [47] Soit une exécution abstraite A. A respecte le modèle de cohérence causale non-spéculative, et nous écrivons $A \in \mathsf{Causal}$, si et seulement si :

C1 (visibilité transitive) La relation de visibilité est transitive.

$$x \xrightarrow{\mathsf{vis}} y \wedge y \xrightarrow{\mathsf{vis}} z \implies x \xrightarrow{\mathsf{vis}} z$$

C2 (visibilité non-spéculative) Une opération future ne peut être visible à une opération passée.

$$y \not\xrightarrow{\mathsf{rb}} x \implies y \not\xrightarrow{\mathsf{vis}} x$$

C3 (ordre linéaire pour chaque pair) Les opérations d'un même pair sont ordonnées par la relation de visibilité.

$$x, y \in A \wedge \mathsf{peer}(x) = \mathsf{peer}(y) \implies (x \xrightarrow{\mathsf{vis}} y \vee y \xrightarrow{\mathsf{vis}} x)$$

Proposition 3.1. *Dans une exécution abstraite A qui respecte le modèle de cohérence causale non-spéculative, la relation de visibilité est égale à la relation retourne-avant pour les opérations d'un même pair.*

$$A \in \mathsf{Causal} \wedge x, y \in A \wedge \mathsf{peer}(x) = \mathsf{peer}(y) \implies (x \xrightarrow{\mathsf{rb}} y \iff x \xrightarrow{\mathsf{vis}} y)$$

Démonstration. La Proposition 3.1 est décomposable en deux implications.

Procédons par contradiction. Soit une exécution abstraite A et deux opérations x et y d'un même pair p telles que x retourne-avant y. Supposons que x n'est pas visible à y. y est visible à x, ou elles n'ont pas de relation de visibilité entre elles. Le premier cas contredit la propriété C2 de visibilité non-spéculative. Le deuxième cas contredit la propriété C3 qui assure une visibilité linéaire pour chaque pair.

La deuxième implication est naturellement déduite de la propriété C2. □

Un protocole peut garantir la cohérence causale avec divers mécanismes [55, 60, 30]. Un protocole naïf pourrait :

— Inclure dans chaque opération x l'ensemble des opérations dont les effets ont été intégrés par l'auteur de x avant l'exécution de x. Cet ensemble correspond au contexte de l'opération.

— Intégrer l'effet d'une opération après l'intégration des effets de l'ensemble des opérations présentes dans son contexte.

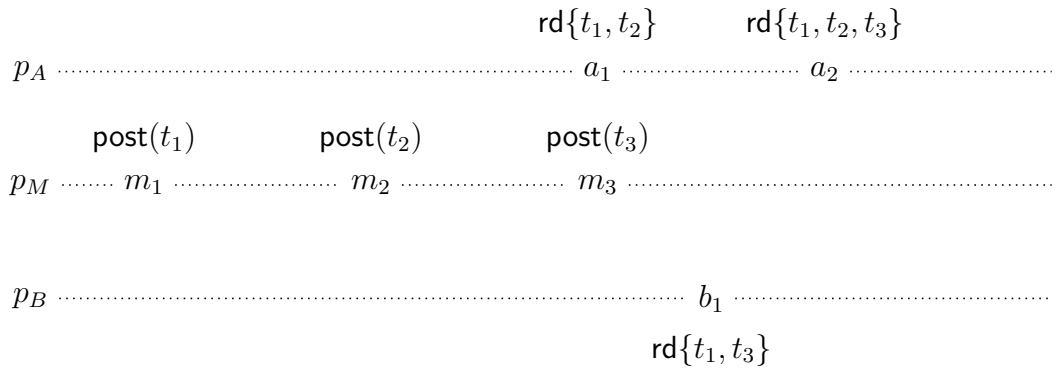
3.2.3 Cohérence *Fork-Join-Causal*

Le modèle de cohérence causale offre des garanties qui facilitent l'implémentation de fonctionnalités dans les applications distribuées. De ce fait, le modèle de cohérence causale est souvent supposé dans les systèmes distribués [28]. Malheureusement, les pairs malintentionnés peuvent compromettre les garanties du modèle de cohérence causale. Un pair malintentionné peut en particulier produire une équivoque. Pour ce faire, il présente

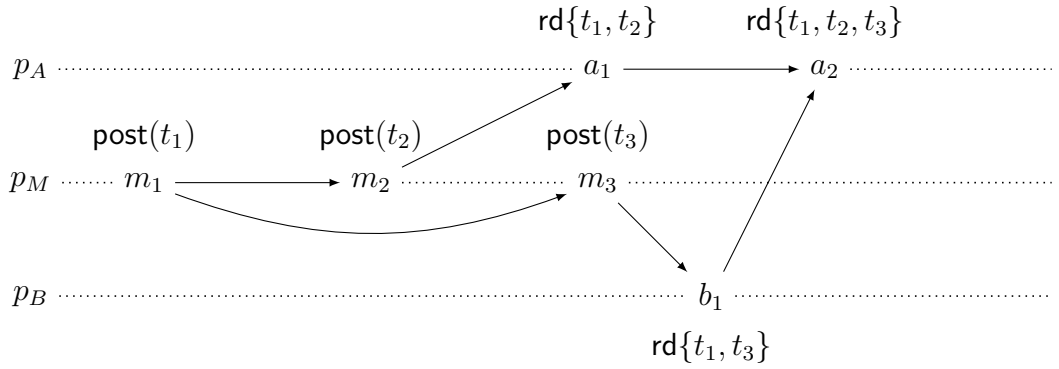
des opérations distinctes comme étant sa n-ième opération à des pairs différents. Par exemple, dans la figure 3.8 le pair malintentionné p_M prétend au pair p_B que sa deuxième opération est m_3, alors qu'il prétend au pair p_A que sa deuxième opération est m_2. p_A lit ainsi les messages textuels t_1 et t_2, alors que p_B lit les messages textuels t_1 et t_3. L'histoire de la figure 3.8 ne peut pas à la fois respecter le modèle de cohérence causale et la spécification du fil de discussion (équation 3.1). L'exécution abstraite de la figure 3.8 respecte la spécification du fil de discussion et viole donc le modèle de cohérence causale.

L'exécution abstraite de la figure 3.8 ne peut pas respecter le modèle de cohérence causale car m_2 ne peut pas être visible à b_1 sans violer la cohérence du contenu répliqué. L'exécution abstraite viole la propriété C3 (visibilité linéaire pour chaque pair) : m_2 n'est pas visible à m_3. Il est démontrable que dans un environnement malintentionné, l'une des trois propriétés de la cohérence causale doit être violée pour pouvoir respecter les deux autres [47].

Le modèle de cohérence Fork-Join-Causal (FJC) [47, 61] propose d'affaiblir la propriété C3 pour tolérer la présence de pairs malintentionnés. Cette dernière est restreinte aux pairs honnêtes. L'exécution abstraite de la figure 3.8 respecte ainsi le modèle de cohérence FJC. La Definition 3.12 définit formellement le modèle de cohérence FJC.



(a) Histoire



(b) Une exécution abstraite qui respecte la spécification du fil de discussion.

FIGURE 3.8 – Exemple d'une équivoque. Les pairs p_A, p_B, et p_M répliquent un fil de discussion. p_M est un pair malintentionné. Il prétend au pair p_A que sa deuxième opération est m_2, alors qu'il prétend au pair p_B que sa deuxième opération est m_3.

Definition 3.12 (Cohérence *Fork-Join-Causal* non-spéculative)**.** [47] Soit une exécution abstraite bien-formée A. A respecte le modèle de cohérence *Fork-Join-Causal (FJC)* non-spéculative, et nous écrivons $A \in \mathsf{FJC}$, si et seulement si :

C1 (visibilité transitive) La relation de visibilité est transitive.

$$x \xrightarrow{\mathsf{vis}} y \wedge y \xrightarrow{\mathsf{vis}} z \implies x \xrightarrow{\mathsf{vis}} z$$

C2 (visibilité non-spéculative) Une opération ne peut être visible à une opération qui a été exécutée avant.

$$y \not\xrightarrow{\mathsf{rb}} x \implies y \not\xrightarrow{\mathsf{vis}} x$$

FJC1 (ordre linéaire pour chaque pair honnête) Les opérations d'un même pair honnête sont ordonnées par la relation de visibilité.

$$x, y \in A \wedge \mathsf{peer}(x) = \mathsf{peer}(y) \wedge \mathsf{peer}(x) \in \mathsf{Honest} \implies (x \xrightarrow{\mathsf{vis}} y \vee y \xrightarrow{\mathsf{vis}} x)$$

Lorsque deux opérations d'un pair malintentionné ne sont pas linéairement ordonnées par la relation de visibilité, nous parlons d'opérations *non-linéaires*[12]. Les opérations non-linéaires forment des *embranchements* (*forks*) [14]. Dans l'exemple présenté, m_2 et m_3 sont deux opérations non-linéaires. Elles forment deux embranchements : $m_1 \longrightarrow m_2 \longrightarrow a_1$ et $m_1 \longrightarrow m_3 \longrightarrow b_1$.

Pour assurer la convergence des pairs honnêtes, les opérations observées par un pair honnête doivent à terme être observées par chaque pair honnête. Par conséquent les embranchements doivent être joints. Les opérations non-linéaires sont donc acceptées par les pairs honnêtes. Seuls les pairs honnêtes ordonnent linéairement leurs opérations. Dans la figure 3.8, le pair p_A joint les deux embranchements avec l'opération a_2.

Proposition 3.2. *Dans une exécution abstraite A qui respecte le modèle de cohérence FJC, la relation de visibilité est égale à la relation retourne-avant pour les opérations d'un même pair honnête.*

$$x, y \in A \wedge \mathsf{peer}(x) = \mathsf{peer}(y) \wedge \mathsf{peer}(x) \in \mathsf{Honest} \implies (x \xrightarrow{\mathsf{rb}} y \iff x \xrightarrow{\mathsf{vis}} y)$$

La preuve de la Proposition 3.2 suit la même structure que la preuve de la Proposition 3.1. « pair » doit être remplacé par « pair honnête » et C3 par FJC1.

Proposition 3.3. *Soit une exécution abstraite qui respecte le modèle de cohérence FJC. Si tous les pairs sont honnêtes, alors l'exécution abstraite respecte également le modèle de cohérence causale.*

$$A \in \mathsf{FJC} \wedge \mathsf{Peers} = \mathsf{Honest} \implies A \in \mathsf{Causal}$$

Démonstration. Si tous les pairs sont honnêtes, alors les propriétés C3 et FJC1 sont équivalentes. □

12. Mahajan et al. [47] parlent d'opérations *non-sérialisées* (*non-serial operations*)

Le modèle de cohérence FJC est plus faible que le modèle de cohérence causale. Les deux modèles partagent les mêmes garanties à l'exception de C3 qui est une spécialisation de FJC1. Toutes les exécutions qui respectent le modèle de cohérence causale respectent également le modèle de cohérence FJC. Une application qui se base sur le modèle de cohérence FJC doit gérer la complexité induite par l'acceptation d'opérations *non-linéaires*, c'est-à-dire d'opérations concurrentes d'un même pair. Il est toutefois possible de présenter une exécution FJC comme une exécution causale en assignant des pairs virtuels aux opérations *non-linéaires* [15]. Ainsi, l'application considère que deux opérations non-linéaires n'ont pas été exécutées par le même pair.

3.2.4 Cohérence *View-Fork-Join-Causal*

Les pairs malintentionnés peuvent réaliser des équivoques. Dans une exécution abstraite, ces équivoques se traduisent par des opérations non-linéaires. Une opération est non-linéaire s'il existe une opération du même auteur qui lui est concurrente. Chaque opération non-linéaire forme un embranchement. Pour préserver la convergence des copies, le modèle de cohérence View-Fork-Join-Causal (VFJC) [47] permet aux pairs d'accepter des opérations non-linéaires. Les pairs peuvent ainsi joindre des embranchements.

Pour certains contenus répliqués, les équivoques peuvent produire des conflits de modification difficiles à identifier et à résoudre. De ce fait, il est souhaitable d'empêcher à terme la réalisation de nouvelles équivoques. Pour ce faire, les pairs qui en produisent doivent être à terme évincés. Leurs opérations futures devraient donc à terme être rejetées.

De manière optimiste, un pair honnête présume que les autres pairs sont honnêtes tant qu'ils ne produisent pas d'équivoques. Un pair reconnaît qu'un pair est malintentionné dès lors qu'il observe deux opérations non-linéaires de ce dernier. Cette observation se traduit par une jonction d'embranchements. Dans l'exécution abstraite A_1 de la figure 3.9, le pair p_A observe les opérations non-linéaires m_2 et m_3 du pair p_M. $p_A \in \mathsf{obs}_{A_1}(m_2) \cap \mathsf{obs}_{A_1}(m_3)$. Il reconnaît ainsi p_M comme malintentionné. Le pair p_B n'a pas encore observé m_2. Il présume que m_3 est une opération linéaire, et donc que p_M est honnête.

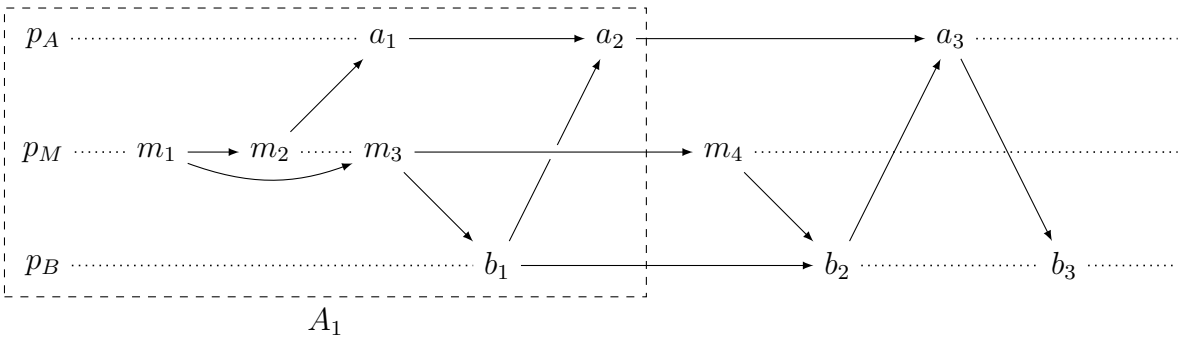


FIGURE 3.9 – Exécutions abstraites qui respectent le modèle de cohérence VFJC.

Pour faciliter la spécification du modèle de cohérence VFJC, nous définissons dans la Definition 3.13 l'ensemble des pairs reconnus malintentionné en une opération. Le tableau 3.3 indique pour chaque opération de la figure 3.9 les pairs qui sont reconnus malintentionnés.

Definition 3.13 (Reconnus malintentionnés)**.** Soit une opération x d'une exécution abstraite A qui respecte le modèle de cohérence Fork-Join-Causal. Les pairs reconnus malintentionnés en x correspondent à ceux qui ont exécuté au moins deux opérations non-linéaires qui sont visibles à x.

$$\mathsf{knownMalicious}_A(x) \stackrel{\text{def}}{=} \{p \mid \exists y, z.\, y \longrightarrow x \wedge z \longrightarrow x \wedge y \| z \wedge p = \mathsf{peer}(y) = \mathsf{peer}(z)\}$$

Opération	$\mathsf{knownMalicious}_A$
a_1	$\emptyset$
a_2	$\{p_M\}$
a_3	$\{p_M\}$
m_1	$\emptyset$
m_2	$\emptyset$
m_3	$\emptyset$
m_4	$\emptyset$
b_1	$\emptyset$
b_2	$\emptyset$
b_3	$\{p_M\}$

TABLE 3.3 – Pairs reconnus malintentionnés dans l'exécution abstraite de la figure 3.9 par le pair qui exécute l'opération considérée.

L'éviction d'un pair reconnu malintentionné ne doit pas menacer la convergence des copies des pairs honnêtes. Dans l'exécution abstraite A_1 de la figure 3.9, p_A reconnaît p_M comme malintentionné. Cependant, p_A ne peut pas systématiquement rejeter les opérations de p_M sans compromettre la convergence des copies des pairs honnêtes. En effet, le pair p_B présume encore p_M comme honnête. Il peut donc encore accepter des opérations futures de p_M. Dans l'exécution abstraite A de la figure 3.9, p_B accepte ainsi m_4. Bien que p_A reconnaît que p_M est malintentionné, il doit accepter m_4 pour ne pas compromettre la convergence de sa copie avec celle de p_B. Nous disons que p_A a indirectement accepté m_4.

Un pair honnête peut rejeter les opérations d'un pair qu'il reconnaît malintentionné si ces dernières ne sont par acceptées par un autre pair. p_A devrait rejeter m_4 tant qu'elle n'est pas acceptée par p_B. Dans l'exemple de la figure 3.9, le pair p_M est évincé une fois que p_A et p_B le reconnaissent tous deux comme malintentionné.

En présence d'autres pairs malintentionnés, il ne suffit pas que les pairs reconnaissent un pair comme malintentionné pour l'évincer du groupe. Un pair malintentionné peut reconnaître un autre pair comme malintentionné et continuer à accepter ses opérations. Dans l'exécution abstraite A_1 de la figure 3.10, les pairs p_A et p_O reconnaissent que p_M est malintentionné. Dans l'exécution abstraite A de la figure 3.10, p_O accepte m_4 alors qu'il a précédemment reconnu que son auteur est malintentionné.

Les pairs honnêtes doivent s'assurer qu'un pair qui reconnaît un autre pair comme malintentionné arrête d'accepter directement les opérations de ce dernier. En d'autres termes, reconnaître qu'un pair est malintentionné devrait être responsabilisant. Si un pair est reconnu malintentionné en une opération x, alors une opération de ce pair ne peut

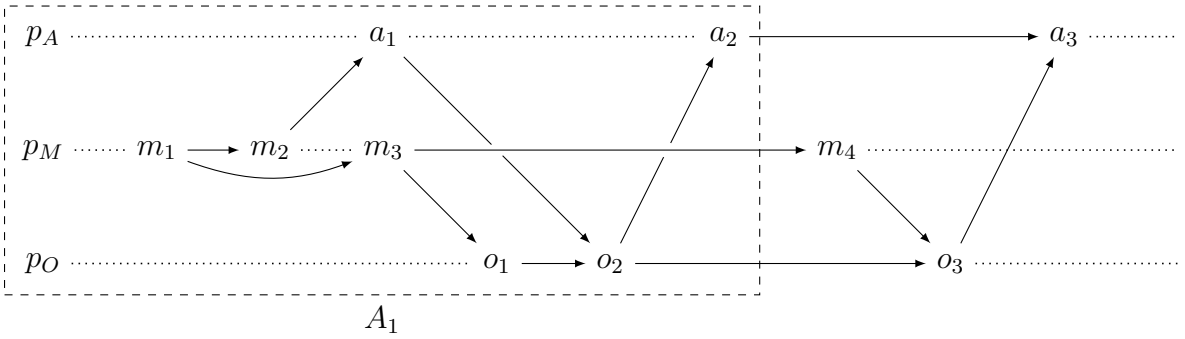


FIGURE 3.10 – Exemple d'une exécution abstraite qui ne respecte pas le modèle de cohérence VFJC. L'exécution abstraite A_1 respecte le modèle de cohérence VFJC.

être immédiatement visible à x. Il s'agit de la propriété VFJC2 du modèle de cohérence View-Fork-Join-Causal (VFJC) que nous présentons dans la Definition 3.14.

Definition 3.14 (Cohérence *View-Fork-Join-Causal*)**.** [47] Soit une exécution abstraite A. A respecte le modèle de cohérence *View-Fork-Join-Causal (VFJC)*, et nous écrivons $A \in \mathsf{VFJC}$, si et seulement si :

VFJC1 (cohérence *Fork-Join-Causal*) L'exécution abstraite A respecte le modèle de cohérence *FJC*.

$$A \in \mathrm{FJC}$$

VFJC2 (partage avec les pairs présumés honnêtes) Si une opération x est immédiatement visible à une opération y, alors aucune des deux opérations n'est exécutée par un pair qui est reconnu malintentionné en y.

$$x \downarrow y \implies \mathsf{peer}(x), \mathsf{peer}(y) \notin \mathsf{knownMalicious}_A(y)$$

Par l'éviction systématique des pairs malintentionnés, le modèle de cohérence VFJC borne le nombre d'embranchements acceptés par les pairs honnêtes. Il accepte uniquement les embranchements qui ne peuvent pas être évités sans compromettre la convergence des copies des pairs. Le nombre maximal d'embranchements acceptés est fonction du nombre de pairs honnêtes et du nombre de pairs malintentionnés.

Le modèle de cohérence VFJC est plus faible que le modèle de cohérence causale. Il est toutefois plus fort que le modèle de cohérence FJC puisqu'il n'accepte qu'un sous-ensemble des embranchements que les pairs malintentionnés peuvent créer.

3.3 Types de données répliquées

La réplication optimiste [12] d'un contenu partagé permet d'interroger et de modifier le contenu à tout moment. Pour ce faire, chaque pair détient sa propre copie du contenu partagé sur-laquelle il exécute ses opérations de modification et ses opérations d'interrogation. Un pair intègre ses modifications avant de les transmettre ultérieurement aux autres pairs qui les intègrent à leur tour. La modification parallèle des copies conduit nécessairement à leur divergence [7]. Les protocoles de réplication optimiste sont responsables de la convergence à terme des copies et plus largement du maintien de la cohérence des copies. Le maintien de la cohérence des copies doit préserver autant que possible l'intention des opérations exécutées.

La conception d'un protocole de réplication n'est pas une tâche aisée [62]. Il doit prendre en compte les aléas du réseau [35] et résoudre les éventuels conflits engendrés par des modifications parallèles et incompatibles. Dans la figure 3.11, les pairs p_A et p_B modifient un ensemble répliqué d'entiers. Alors que p_B ajoute puis supprime un entier dans l'ensemble, le pair p_A ajoute en parallèle ce même entier dans l'ensemble. p_B a l'intention de supprimer l'entier de l'ensemble, alors que p_A a l'intention de l'ajouter à l'ensemble. Il est autant correct que l'ensemble contienne ou ne contienne pas cet entier. La suppression et l'ajout en parallèle d'une même valeur dans un ensemble répliqué sont deux opérations incompatibles.

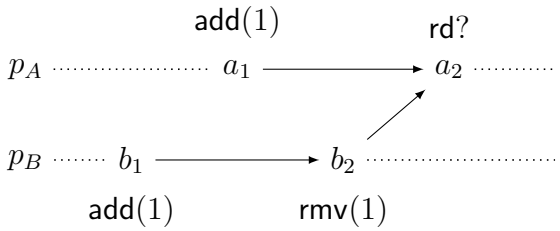


FIGURE 3.11 – Exemple d'exécutions en parallèle d'opérations incompatibles. L'exécution abstraite respecte le modèle de cohérence causale.

Les Conflict-free Replicated Data Types (CRDTs) [28, 24] sont des types de données conçus pour être répliqués et modifiés en parallèle par un ensemble de pairs. Ils encapsulent la complexité des protocoles de réplication optimiste derrière une interface bien définie. Les concepteur·ice·s d'applications peuvent ainsi se concentrer sur la sémantique des contenus répliqués. Nous spécifions un CRDT avec deux modèles [63] : un type abstrait de données répliquées et un schéma de synchronisation. Le type abstrait de données répliquées définit les opérations qui peuvent être exécutées et leur effet lorsqu'elles sont exécutées en parallèle ou en séquence. Nous abordons ce sujet plus en détail dans la sous-section 3.3.1. Le schéma de synchronisation conditionne la manière dont les copies d'un CRDT peuvent se synchroniser. Il repose lui-même sur un modèle réseau. Nous détaillons cet aspect dans la sous-section 3.3.2.

3.3.1 Type abstrait de données répliquées

Pour modifier et consulter un contenu, un pair exécute respectivement des opérations de modification et des opérations d'interrogation. L'exécution des opérations d'interrogation ne modifient pas le contenu. En d'autres termes, leur exécution ou l'absence de leur exécution ne change pas le résultat des interrogations ultérieures. Les opérations qui peuvent être exécutées sur un contenu dépendent du type du contenu. Un type abstrait de données définit les opérations qui peuvent être exécutées sur un ensemble de types de données, ainsi que l'effet de l'exécution de ces opérations. L'ensemble des opérations Op_T et l'ensemble des valeurs de retour des opérations d'interrogation Val_T d'un type abstrait de données T constituent la *signature syntaxique* de T. Cette signature syntaxique est couplée à une sémantique qui détermine les effets de l'exécution des opérations de modification sur les valeurs retournées par l'exécution d'opérations d'interrogation. La sémantique du type abstrait de données est spécifiée par un modèle de cohérence. Elle est donc formulée à l'aide d'un ensemble de propriétés de cohérence.

Prenons l'exemple du type abstrait de données *Ensemble* noté $\mathsf{Set}\langle V\rangle$. V est l'ensemble des valeurs que l'ensemble peut contenir. La figure 3.12 présente une spécification algébrique [64] de ce type. Nous proposons de l'adapter au formalisme utilisé dans ce manuscrit. Les opérations de ce type regroupent une opération de lecture rd, une opération d'ajout $\mathsf{add}(v)$, et une opération de suppression $\mathsf{rmv}(v)$ telle que v est une valeur de l'ensemble V. Les valeurs de retour de l'opération de lecture sont l'ensemble des parties finies de V : $\mathsf{Val}_{\mathsf{Set}\langle V\rangle} = \mathcal{P}_{\mathsf{fin}}(V)$. La sémantique de ce type est précisée par la propriété de cohérence de l'équation 3.3. Chaque valeur incluse dans l'ensemble et retournée par l'exécution d'une opération de lecture est ajoutée par au moins une opération du contexte de l'opération de lecture et n'est pas visible à une opération du contexte de l'opération de lecture qui la supprime.

$$x \in A \wedge \mathsf{call}(x) = \mathsf{rd} \implies \mathsf{rval}(x) = \left\{ v \;\middle|\; \exists y \in \mathsf{ctx}_A(x).\, \mathsf{call}(y) = \mathsf{add}(v) \wedge \nexists z \in \mathsf{ctx}_A(x).\, y \xrightarrow{\mathsf{vis}} z \wedge \mathsf{call}(z) = \mathsf{rmv}(v) \right\} \tag{3.3}$$

Cette propriété de cohérence est insuffisante pour caractériser la sémantique du type Ensemble. En effet, les types usuels supposent des exécutions séquentielles. Ils supposent donc un modèle de cohérence séquentiel dans lequel la relation de visibilité est un ordre total strict.

Le type abstrait de données Ensemble est spécifié pour une exécution séquentielle. Cette sémantique est suffisante aussi longtemps que des modifications parallèles ne surviennent pas. Les CRDTs sont répliqués et peuvent être modifiés en parallèle par plusieurs pairs. Les types abstraits de données qui spécifient les CRDTs doivent donc à la fois spécifier l'effet des opérations et de leur combinaison pour des exécutions séquentielles et des exécutions concurrentes. Ces types abstraits de données sont nommés *types abstraits de données répliquées* [24]. La majorité des CRDTs sont des extensions de types abstraits de données définis pour des exécutions séquentielles. Ils se comportent donc de la même manière lorsqu'ils sont utilisés dans une exécution séquentielle. Il s'agit d'un comportement recherché car il renvoie à une sémantique usuelle pour ses utilisateur·ice·s [63]. Dans ce manuscrit, nous considérons uniquement des CRDTs qui présentent ce comportement.

$S \in \mathsf{Set}\langle V \rangle$

$\mathsf{emp} : S$
$\mathsf{rd} : S \to \mathcal{P}_{\mathsf{fin}}(V)$
$\mathsf{add} : S \times V \to S$
$\mathsf{rmv} : S \times V \to S$

$\mathsf{rd}(\mathsf{emp}) = \emptyset$
$\mathsf{rd}(\mathsf{add}(s, v)) = \{v\} \cup \mathsf{rd}(s)$
$\mathsf{add}(\mathsf{add}(s, v), v) = \mathsf{add}(s, v)$ (add idempotence)
$\mathsf{add}(\mathsf{add}(s, v_1), v_2) = \mathsf{add}(\mathsf{add}(s, v_2), v_1)$ (add commutativity)
$\mathsf{rmv}(\mathsf{emp}, v) = \mathsf{emp}$
$\mathsf{rmv}(\mathsf{add}(s, v), v) = s$
$\mathsf{rmv}(\mathsf{add}(s, v_1), v_2) = \mathsf{add}(\mathsf{rmv}(s, v_2), v_1)$ (add/rmv commutativity)
$\mathsf{rmv}(\mathsf{rmv}(s, v_1), v_2) = \mathsf{rmv}(\mathsf{rmv}(s, v_2), v_1)$ (rmv commutativity)

FIGURE 3.12 – Spécification algébrique du type abstrait de données usuel *Ensemble*. $\mathcal{P}_{\mathsf{fin}}(V)$ désigne l'ensemble des parties finies de V.

La relation de visibilité rend compte des opérations dont l'effet a été intégré. Dans la figure 3.11, les effets des opérations a_1, b_1, et b_2 sont intégrés avant l'exécution de l'opération de lecture a_2. Étant donné que b_1 est visible à b_2, nous pouvons dire que b_2 a pris en compte l'effet de b_1. Elle l'a d'autant pris en compte qu'elle annule l'effet de b_1. L'effet de b_1 devrait donc être intégré avant b_2. En revanche l'effet de a_1 peut être intégré avant b_1, entre b_1 et b_2, ou après b_2. L'ordre d'intégration des effets des opérations peut changer la valeur retournée par b_2. La figure 3.13 présente ces différents ordres d'intégration. Nous constatons que certains ordres d'intégration distincts conduisent à des résultats identiques. Intuitivement, si deux opérations sont commutatives, alors leurs effets peuvent être intégrés dans un ordre quelconque. Dans le cas de l'ensemble, deux ajouts et deux suppressions d'un même élément ou d'éléments distincts sont commutatifs. En revanche l'ajout et la suppression d'un même élément ne sont pas des opérations commutatives. Les opérations sont dites *incompatibles*. Leurs intentions sont incompatibles et conduisent à des conflits de modification si leurs effets sont intégrés dans des ordres distincts. Une sémantique définit un ordre d'intégration des effets des couples d'opérations incompatibles. Puisque cet ordre est déterminé, nous disons que la résolution de conflits est déterministe.

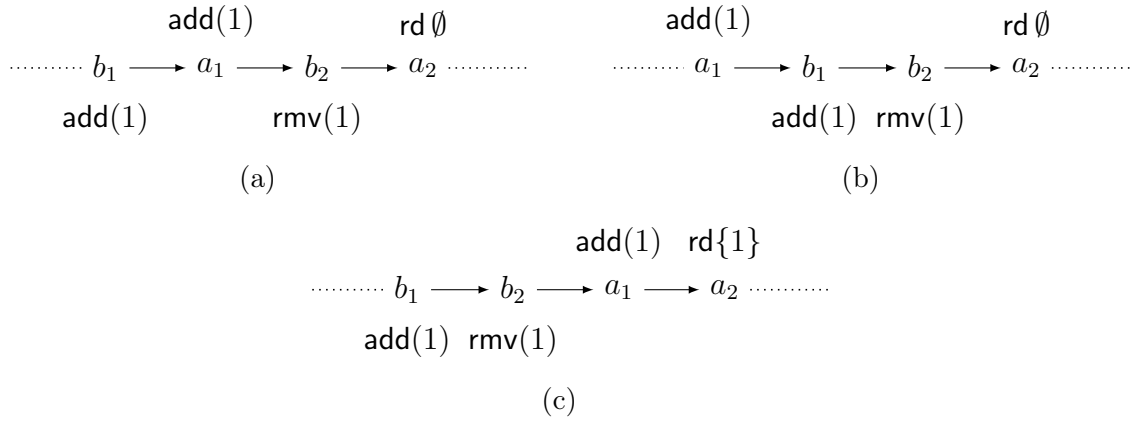


FIGURE 3.13 – Exécutions abstraites qui rend compte des ordres d'intégrations des opérations de l'exécution abstraite de la figure 3.11.

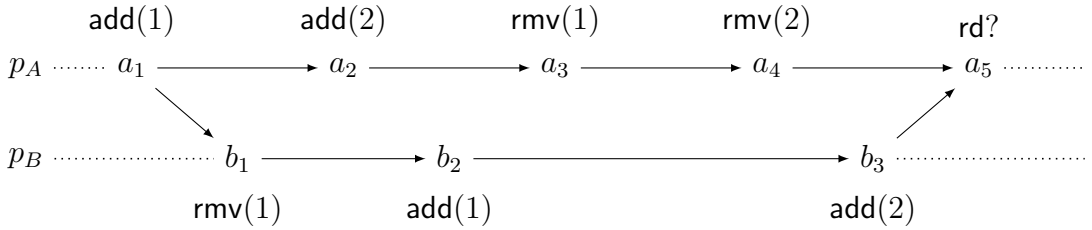
Contrairement aux types abstraits usuels, plusieurs sémantiques peuvent se révéler utiles et dépendre de leur domaine d'application. La littérature a par exemple proposé les sémantiques suivantes pour les ensembles répliqués :

Un Last-Writer-Win Set (LWWSet) [29] spécifie que la dernière opération l'emporte sur les précédentes. Les opérations sont ainsi estampées par un horodatage. Si deux opérations ont le même horodatage, un ordre total strict sur les identifiants des pairs peut être utilisé pour lever l'ambiguïté. Dans la figure 3.14, nous supposons qu'une première opération qui retourne-avant une seconde opération a un horodatage strictement plus petit que l'horodatage de la seconde opération. Pour déterminer la composition de l'ensemble retourné par l'exécution de l'opération de lecture a_5, nous conservons uniquement l'opération avec l'horodatage le plus grand pour chaque valeur ajoutée au moins une fois dans l'ensemble. Pour les valeurs 1 et 2, il s'agit respectivement de l'opération a_3 et b_3. La valeur est présente dans l'ensemble lu si et seulement si l'opération conservée est un ajout. L'ensemble lu est donc un singleton qui contient la valeur 2.

Un Add-Win Set (AWSet) [28] spécifie que l'ajout d'un élément l'emporte toujours sur la suppression de ce même élément en parallèle. Il est aussi connu sous le nom *Observed-remove Set* [28]. Dans la figure 3.14, les ajouts b_2 et b_3 ne précédent pas de suppressions et prennent l'ascendance sur les suppressions concurrentes a_3 et a_4. L'ensemble lu contient donc les valeurs 1 et 2. Cette sémantique est spécifiée par l'équation 3.3.

Un Remove-Win Set (RWSet) spécifie que la suppression d'un élément l'emporte toujours sur l'ajout de cette même valeur en parallèle. Dans la figure 3.14, les suppressions a_3 et a_4 ne précédent pas d'ajouts et prennent l'ascendance sur les ajouts concurrents b_2 et b_3. L'ensemble lu est donc vide. Cette sémantique est spécifiée par l'équation 3.4 :

$$\begin{aligned} x \in A \wedge \mathsf{call}(x) = \mathsf{rd} \implies \mathsf{rval}(x) = \\ \{v \mid \exists y \in \mathsf{ctx}_A(x).\, \mathsf{call}(y) = \mathsf{add}(v) \wedge \nexists z \in \mathsf{ctx}_A(x).\, z \not\leq_{\mathsf{vis}} y \wedge \mathsf{call}(z) = \mathsf{rmv}(v)\} \end{aligned} \tag{3.4}$$



(a)

Sémantique	rval(a_5)
Last-Writer-Win (LWW)	$\{2\}$
Add-Win (AW)	$\{1, 2\}$
Remove-Win (RW)	$\emptyset$
Causal-Length (CL)	$\{1\}$

(b)

FIGURE 3.14 – Illustration des différentes sémantiques d'un ensemble répliqué d'entiers. (a) L'exécution abstraite considérée respecte le modèle de cohérence causale. (b) La valeur de retour de l'opération de lecture a_5 dépend de la sémantique choisie pour l'ensemble répliqué.

Un Causal-Length Set (CLSet) [65] traite les successions d'ajouts et de suppressions d'une même valeur comme des annulations d'ajouts et de suppressions [34]. Si un ajout ou une suppression est parallèle à une opération qui concerne la même valeur, alors l'effet de cette dernière est soit équivalente à la première, ou l'une d'entre elle annule ou prévale sur l'autre. Dans la figure 3.14, les opérations a_3 et b_1 annulent l'effet de l'ajout a_1. Elles sont donc équivalentes. L'ajout b_2 annule l'effet de b_1. Elle annule donc indirectement l'effet des opérations équivalentes à b_1, à savoir a_3. En suivant le même principe on constate que a_2 et b_3 sont équivalentes. a_4 annule l'effet de a_2 et b_3. L'ensemble lu est donc un singleton qui contient la valeur 1.

Alors que les types abstraits usuels supposent le respect d'un modèle de cohérence séquentiel, les types abstraits de données répliquées respectent des modèles de cohérence plus faibles. Les CRDTs respectent par conception la propriété de convergence forte. Cette dernière est impliquée par leur sémantique. En fonction du schéma de synchronisation choisi, les CRDTs peuvent respecter des modèles de cohérence plus fort.

Les types abstraits de données répliquées permettent aux utilisateur·ice·s de se concentrer sur le comportement des contenus répliqués. Les CRDTs s'appuie sur cette abstraction pour encapsuler les mécanismes de résolution de conflits. La réplication optimiste d'un contenu partagé requiert la synchronisation des copies du contenu. Les CRDTs proposent différentes stratégies de synchronisation que nous présentons dans la sous-section suivante.

3.3.2 Schéma de synchronisation

Afin de converger, les pairs doivent intégrer l'ensemble des modifications effectuées sur les copies du contenu partagé. Pour ce faire, les pairs s'échangent leurs modifications sous forme de messages. Lorsqu'un pair exécute une opération, il génère un message qu'il intègre à sa copie et qu'il transmet aux autres pairs. Les pairs intègrent à terme les messages qu'ils reçoivent.

La forme des messages conditionne la manière de synchroniser les copies du contenu partagé. Il existe deux principales familles de CRDTs : les CRDTs synchronisés par opérations, et les CRDTs synchronisés par états. Nous présentons chaque famille de CRDTs et nous les illustrons à l'aide d'un ensemble répliqué. L'ensemble répliqué choisi respecte la sémantique *add-win* présentée dans la sous-section 3.3.1.

Quel que soit le schéma de synchronisation choisi, les implémentations d'ensembles répliqués que nous présentons se basent sur une idée commune. Chaque ajout d'une valeur dans l'ensemble associe à cette valeur un identifiant unique. Si la valeur est ajoutée plusieurs fois dans l'ensemble, elle est donc associée à plusieurs identifiants. Une suppression ne supprime pas directement une valeur mais supprime définitivement un ensemble d'identifiants qui lui est associé. Une valeur est présente dans l'ensemble si elle est associée à au moins un identifiant. L'utilisation d'identifiants uniques est courant dans les implémentations de CRDTs [13, 18]. Ces identifiants sont généralement des couples dont le premier élément est l'identifiant unique du pair, et le second élément est un entier naturel qui est incrémenté avant chaque génération d'un nouvel identifiant. Ce second élément est communément nommé *nombre séquentiel*. Ces couples sont dénommés *dots* [13].

Pour décrire l'implémentation de ces ensembles répliqués, nous introduisons un formalisme inspiré des travaux de Baquero et al. [13]. Notre formalisme nous permet de décrire de manière unifiée les CRDTs synchronisés par opérations et les CRDTs synchronisés par états. Une fois initialisé, un CRDT peut être soumis aux événements d'entrés suivants :

— L'exécution d'une opération d'interrogation

— L'exécution d'une opération de modification

— L'intégration d'un message d'un autre pair

Lorsqu'un pair exécute une opération d'interrogation, il évalue la valeur de retour de cette interrogation à l'aide de la fonction eval. L'exécution d'une opération de modification se déroule en trois étapes. Un message est d'abord généré à l'aide de la fonction prepare. Une fois généré, le message est intégré à la copie du pair à l'aide de la fonction integrate. Finalement, le message est laissé à la charge d'un protocole de passage de messages qui peut transmettre immédiatement ou ultérieurement le message aux autres pairs. L'intégration d'un message d'un autre pair s'effectue simplement à l'aide de la fonction integrate.

L'implémentation des fonctions eval, prepare, et integrate, ainsi que la spécification de l'ensemble des états suffisent à décrire l'implémentation d'un CRDT. Le tableau 3.4 donne la signature et la sémantique de ces fonctions. Dans les paragraphes suivants nous détaillons les différences entre les différentes familles de CRDTs.

$\mathsf{eval}_i(\sigma, x)$	Retourne la valeur de retour de l'exécution de l'opération d'interrogation x par le pair p_i sur l'état σ
$\mathsf{prepare}_i(\sigma, x)$	Retourne un message généré par l'exécution de l'opération de modification x par le pair p_i sur l'état σ
$\mathsf{integrate}_i(\sigma, m)$	Retourne la mise-à-jour de l'état σ qui correspond à σ auquel a été intégré le message m sur le pair p_i

TABLE 3.4 – Fonctions qui permettent d'implémenter un CRDT.

Les CRDTs synchronisés par opérations [66, 13] représentent les modifications effectuées sur une copie par des opérations intégrables. Il est important de distinguer les opérations exécutées et les opérations intégrables. L'exécution d'une opération de modification donne lieu à la génération d'un message qui encapsule une opération intégrable. Par abus de langage nous pouvons parler d'opérations lorsque l'ambiguïté n'est pas possible. Ces opérations peuvent être intégrées sur toutes les copies sans générer de conflits. Toutefois elles ne peuvent pas être intégrées dans des ordres quelconques. Il existe des relations de dépendances entre les opérations intégrables.

Les CRDTs synchronisés par opérations exigent un protocole de passage fiable de message : chaque message doit être livré exactement une fois. Cette famille de CRDTs requiert en général un ordre non-arbitraire de livraison des messages. Ils supposent souvent une livraison causale des messages car elle garantit une cohérence causale des copies et elle simplifie leur implémentation. En d'autres termes, si un pair intègre un message x avant l'intégration d'un de ses propres messages y, alors x doit toujours être intégré avant y quel que soit la copie. En pratique, la livraison causale d'opérations est difficile et peut entraîner des ralentissements de propagation des opérations dans l'ensemble du système [31].

$$\mathsf{OpAWSet}\langle V\rangle \stackrel{\text{def}}{=} \mathcal{P}_{\mathsf{fin}}(V \times \mathbb{I} \times \mathbb{N}^*) \times \mathbb{N}_0 \tag{3.5}$$

$$\mathsf{eval}_i(\langle t, n\rangle, \mathsf{rd}) \stackrel{\text{def}}{=} \{v \mid \langle v, \_, \_\rangle \in t\} \tag{3.6}$$

$$\mathsf{prepare}_i(\langle t, n\rangle, \mathsf{add}(v)) \stackrel{\text{def}}{=} \langle \mathsf{add}, \langle v, i, n+1\rangle\rangle \tag{3.7}$$

$$\mathsf{prepare}_i(\langle t, n\rangle, \mathsf{rmv}(v)) \stackrel{\text{def}}{=} \langle \mathsf{rmv}, \{\langle v, \_, \_\rangle \in t\}\rangle \tag{3.8}$$

$$\begin{aligned}&\mathsf{integrate}_i(\langle m, n\rangle, \langle \mathsf{add}, \langle v, j, n''\rangle\rangle) \stackrel{\text{def}}{=} \langle t \cup \{\langle v, j, n''\rangle\}, n'\rangle \\ &\qquad \textbf{where } n' \stackrel{\text{def}}{=} \begin{cases} n'' & \textbf{if } i = j \\ n \end{cases}\end{aligned} \tag{3.9}$$

$$\mathsf{integrate}_i(\langle t, n\rangle, \langle \mathsf{rmv}, x\rangle) \stackrel{\text{def}}{=} \langle t - x, n\rangle \tag{3.10}$$

FIGURE 3.15 – Implémentation d'un ensemble répliqué synchronisé par opérations qui respecte la sémantique *Add-Win*. La suppression d'une valeur doit être intégrée après l'intégration de son ajout.

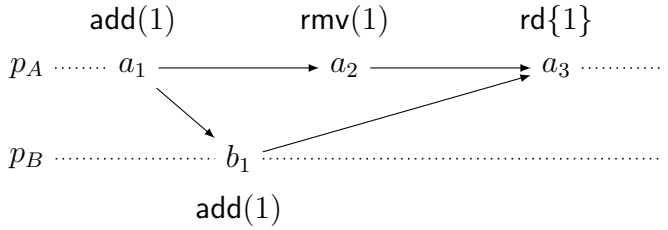


FIGURE 3.16 – Exécution abstraite qui respecte le modèle de cohérence causale. Le contenu répliqué est un ensemble qui respecte la sémantique *Add-Win*.

La figure 3.15 propose une implémentation d'un ensemble répliqué synchronisé par opérations. L'équation 3.5 précise l'ensemble des états du CRDT. Un état est défini comme un couple $\langle t, n\rangle$ où t contient l'ensemble des valeurs associées à leur identifiant unique et n est un nombre séquentiel utilisé pour générer les identifiants uniques. L'association d'une valeur v et d'un identifiant $\langle i, n'\rangle$ est représentée par un triplet $\langle v, i, n'\rangle$ où i est pris dans l'ensemble $\mathbb{I}$ des identifiants des pairs et n' est un entier naturel non-nul ($n' \in \mathbb{N}^*$). L'évaluation d'une opération de lecture retourne l'ensemble des valeurs enregistrées dans au moins un triplet (équation 3.6). L'opération intégrable d'ajout d'une valeur correspond à un couple constitué de l'étiquette add et du triplet composé de la valeur et d'un nouvel identifiant unique (équation 3.7). L'identifiant unique est formé par l'identifiant du pair qui génère l'opération intégrable et de son nombre séquentiel incrémenté d'une unité. L'intégration d'une opération d'ajout consiste à ajouter le nouveau triplet et à incrémenter le nombre séquentiel si l'opération a été générée par le pair actuel (équation 3.9). L'opération intégrable de suppression contient l'ensemble des triplets à supprimer et est estampillé par l'étiquette rmv (équation 3.8). L'intégration de l'opération de suppression supprime les triplets qu'elle contient (équation 3.10).

Nous prenons l'exécution abstraite de la figure 3.16 pour illustrer la réplication d'un ensemble qui utilise l'implémentation de la figure 3.15. Le pair p_A exécute une première opération a_1 qui ajoute la valeur 1 à l'ensemble répliqué. Il transmet sa modification au pair p_B. p_B intègre la modification de p_A et exécute ensuite l'opération b_1 qui ajoute de nouveau la valeur 1 dans l'ensemble. Pendant ce temps, p_A exécute l'opération a_2 qui supprime la valeur 1. p_A reçoit et intègre la modification de p_B. p_A termine par l'exécution de l'opération de lecture a_3. La figure 3.17 décrit les états successifs de la copie du pair p_A. Un triplet $\langle v, i, n\rangle$ et un identifiant $\langle i, n\rangle$ sont respectivement notés $\langle v, i_n\rangle$ et i_n. Les arcs dirigés indiquent des transitions d'états. Ils sont étiquetés à gauche des événements qui produisent les transitions d'états. Lorsque l'événement correspond à l'exécution d'une opération de modification ou d'une opération d'interrogation, les arcs dirigés sont respectivement étiquetés à droite par le message généré et la valeur de retour.

L'exécution d'une opération de suppression d'une valeur v forme une opération intégrable qui contient l'ensemble des identifiants associés à v. Si une opération d'ajout de v est exécutée en parallèle, un nouvel identifiant est associé à la valeur. Cet identifiant n'est pas inclus dans la suppression. Quel que soit l'ordre d'intégration de ces deux opérations, la valeur v sera présente dans l'ensemble. L'implémentation décrite respecte donc la sémantique *Add-Win*.

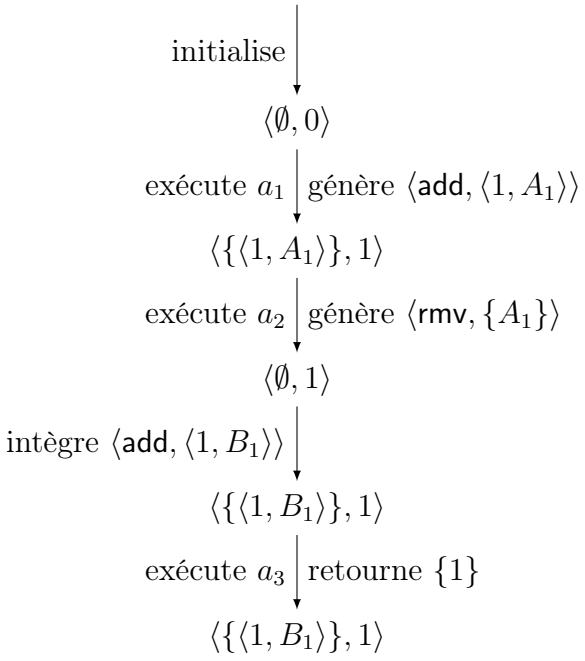


FIGURE 3.17 – États successifs de la copie du pair p_A de l'exécution abstraite de la figure 3.16. L'ensemble répliqué utilise l'implémentation de la figure 3.15.

Les CRDTs synchronisés par états transmettent directement l'état mis à jour au reste des pairs. L'ensemble des messages et l'ensemble des états du CRDT sont donc confondus. Lorsqu'un pair reçoit l'état mis à jour d'un autre pair, il le fusionne avec l'état actuel de sa copie. La fonction d'intégration integrate permet donc de fusionner deux états. Les états peuvent être fusionnés dans un ordre arbitraire et plusieurs fois sans compromettre la convergence des copies. La fonction d'intégration est par conséquent associative, commutative, et idempotente.

Les états sont partiellement ordonnés de manière à ce que la fusion de deux états produisent toujours un état plus grand que les états fusionnés ou égal à l'un des états fusionnés. L'état obtenu est unique et minimal. Il s'agit de la borne supérieure des états fusionnés. Ainsi un état fusionné avec lui-même donne l'état lui-même. La fusion d'un état σ_1 avec un état supérieur σ_2 donne σ_2. Ces propriétés forment une structure algébrique connue. En effet, l'ensemble des états équipé de la fonction d'intégration forme un sup-demi-treillis [67]. Chaque modification enfle l'état actuel pour obtenir un état qui lui est supérieur ou égal. Ainsi si l'appel $\mathsf{prepare}_i(\sigma_1, \_)$ donne l'état σ_2, alors σ_2 est supérieur ou égal à σ_2 et leur fusion donne σ_2.

La propriété d'idempotence de la fusion et la propriété d'inflation d'une modification rendent les CRDTs synchronisés par états particulièrement adaptés aux systèmes de passage de messages non-fiables. Elles permettent d'assurer la convergence à terme en dépit de la perte, du retard, de la duplication, ou du réordonnement de messages. Si des pairs ne reçoivent pas un état d'un pair, il suffit qu'ils reçoivent un état ultérieur de ce pair pour obtenir les modifications incluses dans l'état qu'ils n'ont pas reçu. Leur manière de se synchroniser permet également au CRDT d'être causalement cohérent. Sur ce point, ils

$$\mathsf{StateAWSet}\langle V\rangle \stackrel{\text{def}}{=} \mathcal{P}_{\mathsf{fin}}(V \times \mathbb{I} \times \mathbb{N}^*) \times \mathcal{P}_{\mathsf{fin}}(\mathbb{I} \times \mathbb{N}^*) \tag{3.11}$$

$$\mathsf{eval}_i(\langle t, c\rangle, \mathsf{rd}) \stackrel{\text{def}}{=} \{v \mid \langle v, \_, \_\rangle \in t\} \tag{3.12}$$

$$\begin{aligned}\mathsf{prepare}_i(\langle t, c\rangle, \mathsf{add}(v)) &\stackrel{\text{def}}{=} \langle t \cup \{\langle v, i, n\rangle\}, c \cup \langle i, n\rangle\rangle \\ &\textbf{where } n \stackrel{\text{def}}{=} 1 + \max(\{0, n \mid \langle i, n\rangle \in c\})\end{aligned} \tag{3.13}$$

$$\mathsf{prepare}_i(\langle t, c\rangle, \mathsf{rmv}(v)) \stackrel{\text{def}}{=} \langle t - \{\langle v, \_, \_\rangle \in t\}, c\rangle \tag{3.14}$$

$$\begin{aligned}\mathsf{integrate}_i(\langle t, c\rangle, \langle t', c'\rangle) &\stackrel{\text{def}}{=} \langle t'', c \cup c'\rangle \\ &\textbf{where } t'' \stackrel{\text{def}}{=} (t \cap t') \\ &\qquad \cup \{\langle v, j, n\rangle \in t \mid \langle j, n\rangle \notin c'\} \\ &\qquad \cup \{\langle v', j', n'\rangle \in t' \mid \langle j', n'\rangle \notin c\}\end{aligned} \tag{3.15}$$

FIGURE 3.18 – Implémentation d'un ensemble répliqué synchronisé par états qui respecte la sémantique *Add-Win*.

sont plus avantageux que les CRDTs synchronisés par opérations. Lorsque la complexité spatiale d'un état est importante, les CRDTs synchronisés par états peuvent induire un coût de transmission non-négligeable. Le coût peut être si élevé qu'un CRDT synchronisé par opérations est préférable.

La figure 3.18 présente une implémentation d'un ensemble répliqué synchronisé par états. L'équation 3.11 précise l'ensemble des états du CRDT. Un état consiste en un couple $\langle t, c\rangle$ où t est l'ensemble des valeurs associées à leur identifiant, et c est l'ensemble des identifiants générés par les pairs. Les associations de valeurs et d'identifiants sont représentées de la même manière que celles de l'ensemble répliqué synchronisé par opérations. c est le *contexte causal* [18] du CRDT. Il permet d'éviter d'ajouter ou de ré-ajouter une association qui a été supprimée. L'évaluation d'une opération de lecture retourne l'ensemble des valeurs enregistrées dans au moins un triplet (équation 3.12). L'état généré par l'ajout d'une valeur correspond à l'ajout d'un nouveau triplet et de son identifiant au contexte causal (équation 3.13). Le nouvel identifiant est formé par l'identifiant du pair qui exécute l'opération et le nombre séquentiel de ce pair incrémenté d'une unité. Le nombre séquentiel d'un pair est obtenu à partir du contexte causal. L'état généré par la suppression d'une valeur ne modifie pas le contexte causal et retire les triplets correspondants (équation 3.14). Le contexte causal résultant de la fusion de deux états est l'union des contextes causaux des deux états (équation 3.15). La formation de l'ensemble des triplets est un peu plus difficile. Nous devons veiller à ne pas ajouter des triplets supprimés et à ajouter les triplets qui ne l'ont pas encore été dans l'un des états. Un triplet est supprimé d'un état si son identifiant est présent dans son contexte causal. Inversement un triplet n'a pas encore été ajouté dans un état si son identifiant est absent du contexte causal de l'état. L'ensemble des triplets contient donc les triplets présents dans chaque état et les triplets de chaque état dont l'identifiant est absent du contexte causal de l'autre état. En pratique le contexte causal peut être représenté par un vecteur de versions [68, 69].

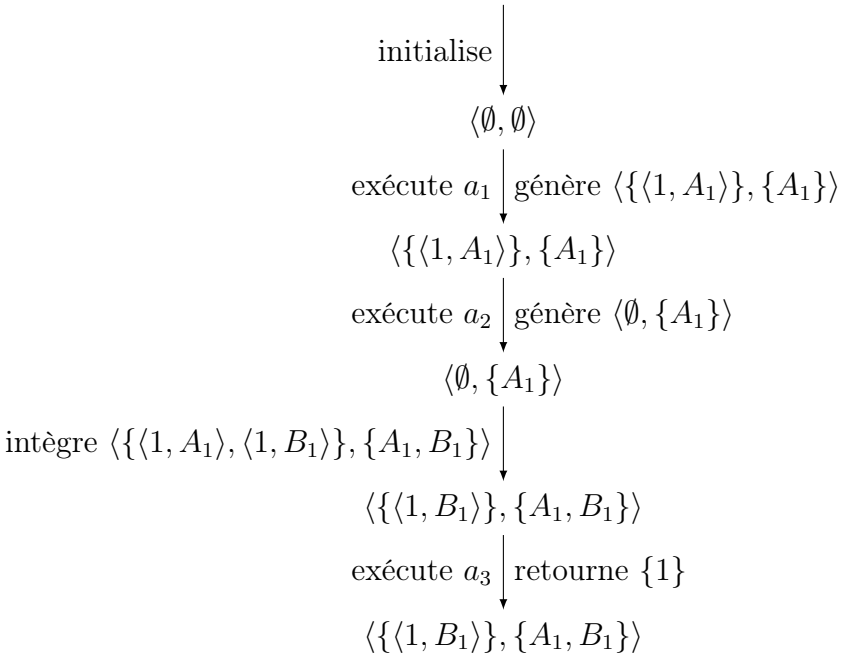


FIGURE 3.19 – États successifs de la copie du pair p_A de l'exécution abstraite de la figure 3.16. L'ensemble répliqué utilise l'implémentation de la figure 3.18.

Nous prenons l'exécution abstraite de la figure 3.16 pour illustrer la réplication d'un ensemble qui utilise l'implémentation de la figure 3.18. La figure 3.19 décrit les états successifs de la copie du pair p_A. La figure 3.20 présente le sup-demi-treillis des états de cette exécution. Les traits épaissis représentent les transitions d'états du pair p_A. La fusion de deux états correspond à leur borne supérieure dans le sup-demi treillis. Ainsi la fusion des états ⟨∅, {A_1}⟩ et ⟨{⟨1, A_1⟩, ⟨1, B_1⟩}, {A_1, B_1}⟩ produit l'état ⟨{⟨1, B_1⟩}, {A_1, B_1}⟩.

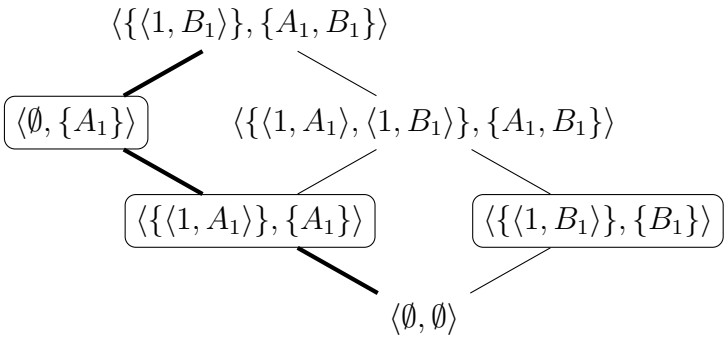


FIGURE 3.20 – Diagramme de Hasse du sup-demi-treillis des états de l'ensemble synchronisé par états de l'exécution de la figure 3.16. Les états entourés sont irréductibles.

Les CRDTs synchronisés par différences [18] sont une sous-famille des CRDTs synchronisés par états. Ils permettent la conception de mécanismes de synchronisation plus efficaces [70]. Ils reposent sur l'idée de décomposer un état en un ensemble d'états irréductibles. Un état est irréductible si et seulement si il ne peut pas être obtenu par la fusion de deux états distincts et s'il ne correspond pas à un état minimal (un état qui est inférieur à tout autre état). Dans la figure 3.20, les états irréductibles sont entourés. L'état minimal est $\langle \emptyset, \emptyset \rangle$. Un état décomposé peut être obtenu par la fusion de l'ensemble de ses états irréductibles. Par exemple, l'état $\langle \{\langle 1, B_1 \rangle\}, \{A_1, B_1\} \rangle$ est obtenu par la fusion des deux états irréductibles $\langle \emptyset, \{A_1\} \rangle$ et $\langle \{\langle 1, B_1 \rangle\}, \{B_1\} \rangle$. Au lieu de transmettre aux autres pairs un état mis à jour dans son intégralité, un pair peut transmettre la différence entre un état mis à jour et sa version précédente. Dans la littérature, plusieurs mécanismes de synchronisation par différences d'états tirent avantage de cette observation. Certains d'entre eux permettent d'assurer une cohérence causale avec un surcoût plus faible que les mécanismes de synchronisation par état complet [70]. La flexibilité de cette approche permet la conception de mécanismes de synchronisation variés et est donc adaptée à un large éventail d'applications.

Comparé aux CRDT synchronisés par états complets, l'appel à la fonction $\mathsf{prepare}$ ne retourne pas l'état mis-à-jour mais une différence d'état. Bien que ce ne soit pas nécessaire, nous supposons que cette différence est minimale. En d'autres termes nous ne pouvons trouver un autre état plus petit qui produit la même inflation. Cette propriété n'est pas requise dans la définition originale des CRDT synchronisés par différences d'états. Elle a été introduire ultérieurement [70].

La figure 3.21 présente l'implémentation d'un ensemble répliqué synchronisé par différences d'états. Elle est globalement identique à l'implémentation présentée dans la figure 3.18. Seuls les messages générés sont distincts. Les messages générés sont des états irréductibles.

$$\mathsf{DeltaAWSet}\langle V \rangle \stackrel{\text{def}}{=} \mathcal{P}_{\mathsf{fin}}(V \times \mathbb{I} \times \mathbb{N}^*) \times \mathcal{P}_{\mathsf{fin}}(\mathbb{I} \times \mathbb{N}^*) \tag{3.16}$$

$$\mathsf{eval}_i(\langle t, c \rangle, \mathsf{rd}) \stackrel{\text{def}}{=} \{v \mid \langle v, \_, \_ \rangle \in t\} \tag{3.17}$$

$$\begin{aligned} &\mathsf{prepare}_i(\langle t, c \rangle, \mathsf{add}(v)) \stackrel{\text{def}}{=} \langle \langle v, i, n \rangle, \{i, n\} \rangle \\ &\qquad \textbf{where } n \stackrel{\text{def}}{=} 1 + \max(\{0, n \mid \langle i, n \rangle \in c\}) \end{aligned} \tag{3.18}$$

$$\mathsf{prepare}_i(\langle t, c \rangle, \mathsf{rmv}(v)) \stackrel{\text{def}}{=} \langle \emptyset, \{\langle i, n \rangle \mid \langle v, i, n \rangle \in t\} \rangle \tag{3.19}$$

$$\begin{aligned} &\mathsf{integrate}_i(\langle t, c \rangle, \langle t', c' \rangle) \stackrel{\text{def}}{=} \langle t'', c \cup c' \rangle \\ &\qquad \textbf{where } t'' \stackrel{\text{def}}{=} (t \cap t') \\ &\qquad\qquad \cup \{\langle v, j, n \rangle \in t \mid \langle j, n \rangle \notin c'\} \\ &\qquad\qquad \cup \{\langle v', j', n' \rangle \in t' \mid \langle j', n' \rangle \notin c\} \end{aligned} \tag{3.20}$$

FIGURE 3.21 – Implémentation d'un ensemble répliqué synchronisé par différences d'états qui respecte la sémantique *Add-Win*.

Nous prenons de nouveau l'exécution abstraite de la figure 3.16 pour illustrer la réplication d'un ensemble qui utilise l'implémentation de la figure 3.21. Le pair p_A a exactement les mêmes transitions d'états que l'ensemble synchronisé par états complets. Il génère les mêmes messages qui correspondent à des états irréductibles. La figure 3.19 représente donc les états successifs de la copie du pair p_A. Dans cet exemple, le pair p_B transmet à p_A l'état $\langle\{\langle 1, A_1\rangle, \langle 1, B_1\rangle\}, \{A_1, B_1\}\rangle$. La synchronisation par différences offre à p_B plus de liberté. Un autre protocole de synchronisation pourrait envoyer l'état irréductible $\langle\{\langle 1, B_1\rangle\}, \{B_1\}\rangle$.

Les CRDTs synchronisés par différences d'états conservent les avantages des CRDTs synchronisés par états tout en corrigeant leurs principaux défauts. Avec un mécanisme de synchronisation adéquate, ils peuvent supporter la perte, les duplications, et l'intégration dans un ordre arbitraire des messages. Ils ont la capacité de réduire le coût de transmission puisque seules des différences d'états peuvent être transmises.

Nous avons présenté les deux principales familles de CRDTs : les CRDTs synchronisés par opérations et les CRDTs synchronisés par états. Les CRDTs synchronisés par opérations exige un mécanisme de passage de message fiable et requiert l'intégration de certains messages dans un ordre spécifique au CRDT. Les CRDTs synchronisés par états sont moins exigeants et supportent les mécanismes de passage de message non-fiable avec une intégration de tous les messages dans un ordre quelconque. Cependant, la taille des messages de certains CRDTs synchronisés par états peut être impraticable. Les CRDTs synchronisés par différences d'états corrigent ce problème. Au lieu de transmettre l'intégralité de l'état mis à jour, ils peuvent transmettre des différences d'états. Dans le chapitre 5 nous proposons un protocole synchronisé par différences d'états adapté à l'édition collaborative de texte.

Chapitre 4

Journaux infalsifiables tronqués

Sommaire

Dans le chapitre 2 nous avons vu que la réplication optimiste d'un contenu et la modification concurrente de ce contenu conduisent à la divergence des copies du contenu. Les protocoles de réplication sont responsables de la convergence des copies. La convergence des copies des pairs est une propriété essentielle d'un système collaboratif. Si les copies des pairs ne sont pas capables de converger, les pairs ne disposent pas du même contexte pour collaborer. Ce qui peut conduire à des incompréhensions et à l'échec de la collaboration.

Un individu peut trouver un intérêt à perturber, voire compromettre le succès d'une collaboration. Nous nous intéressons en particulier à un individu qui cherche à compromettre la convergence des copies. Cet individu est l'adversaire de la collaboration. Pour atteindre ses objectifs, l'adversaire contrôle le réseau et un ensemble de pairs que nous qualifions de *malintentionnés*. Les pairs qui ne sont pas sous le contrôle de l'adversaire sont qualifiés d'*honnêtes*.

La présence de pairs malintentionnés représente un défi. Ils peuvent exposer des comportements malintentionnés difficiles à détecter et déjouer. Ils peuvent en particulier produire des *équivoques*. Une équivoque consiste à transmettre à différents pairs des modifications distinctes qui sont perçues comme identiques par le protocole de réplication. Les copies des pairs honnêtes divergent alors de manière permanente sans que les pairs honnêtes s'en rendent compte.

Pour détecter les équivoques et intégrer à leur copie l'ensemble des modifications effectuées sur le contenu, les pairs peuvent répliquer un *journal infalsifiable*. Chaque pair possède un journal dans lequel il enregistre l'ensemble des modifications qu'il intègre à sa copie du contenu partagé. Une modification est accompagnée d'une signature cryptographique qui permet de s'assurer de son authenticité. Un pair malintentionné ne peut donc pas prétendre qu'une de ses modifications a été effectuée par un pair honnête. Un journal infalsifiable contient uniquement des modifications authentiques.

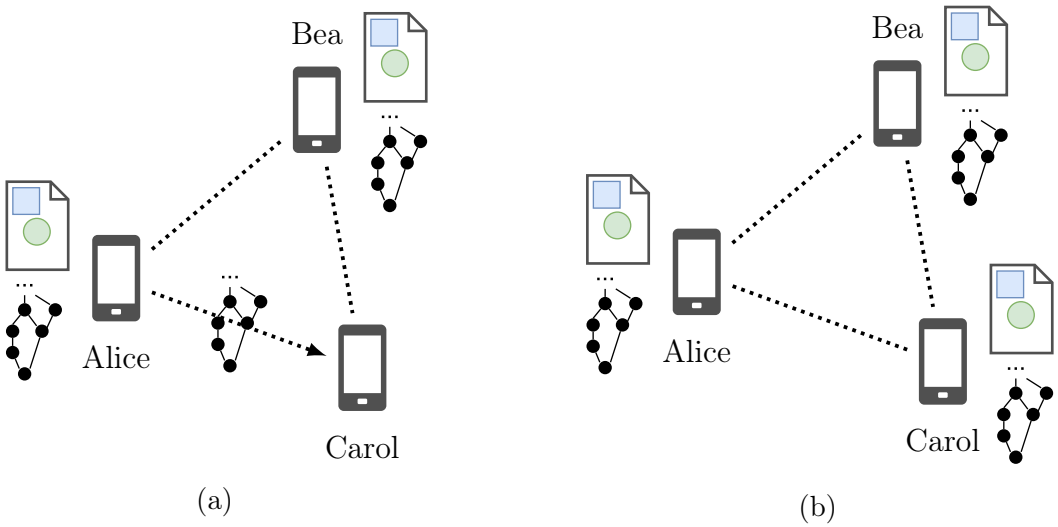


FIGURE 4.1 – Alice et Bea répliquent de manière optimiste un dessin. Elles utilisent un journal infalsifiable pour assurer la convergence des copies des pairs honnêtes. Carol veut rejoindre la collaboration. (a) Alice envoie son journal à Carol. (b) Carol vérifie l'authenticité du journal et construit le dessin à partir du journal.

Les pairs doivent préserver l'intégralité de leur journal pour (i) détecter les équivoques et (ii) permettre à un pair de rejoindre la collaboration. Un pair qui rejoint la collaboration a en effet besoin du journal pour obtenir l'état actuel du contenu partagé. Il obtient cet état en intégrant sur sa copie vierge du contenu partagé chaque modification enregistrée dans le journal. La figure 4.1 illustre la transmission d'un journal infalsifiable à un pair qui rejoint la collaboration. La préservation de l'intégralité du journal engendre plusieurs problèmes :

Passage à l'échelle. Au fur et à mesure de la progression de la collaboration, les journaux contiennent de plus en plus de modifications. La préservation de l'intégralité d'un journal et leur transmission aux pairs qui rejoignent la collaboration sont donc de plus en plus coûteuses.

Vie Privée. La préservation du journal dans son intégralité expose l'historique de la collaboration depuis son commencement. Bien que certaines applications requièrent cet historique pour fournir des fonctionnalités, telles que la visualisation interactive de l'évolution d'un contenu, il peut être intéressant d'offrir plus de liberté sur quelles données doivent être conservées et quelles données doivent être supprimées.

Pour répondre à ces deux problématiques, nous effectuons deux observations :

— Toutes les modifications ne sont pas forcément nécessaires à la détection d'équivoques.

— L'occupation mémoire de la copie du contenu partagé est généralement plus faible que celle du journal.

Nous proposons ainsi deux mécanismes :

Troncature du journal. Au lieu de préserver l'intégralité du journal, seules les modifications nécessaires à la détection d'équivoques pourraient être conservées. Pour supprimer une modification du journal, nous devons nous assurer qu'elle n'est pas nécessaire à la détection d'éventuelles équivoques futures. Pour ce faire, nous utilisons des journaux infalsifiables dans lesquels chaque modification déclare d'autres modifications comme dépendances. Les dépendances sont déclarées de manière à résister aux équivoques. En d'autres termes, si un pair malintentionné effectue une équivoque en générant deux modifications x et y et qu'une modification z déclare dépendre de x, alors nous considérons qu'il n'est pas possible de prétendre que z dépend de y. Les modifications et leurs dépendances forment ainsi un graphe dirigé sans cycle. Nous contraignons l'acceptation de modifications dans un journal de manière à ce que les modifications acceptées ultérieurement partagent un ensemble croissant de dépendances. Nous déterminons cet ensemble à l'aide du concept de *stabilité*. Au sein d'un journal, une modification devient *stable* une fois que toutes les modifications acceptées ultérieurement dans le journal dépendent de cette modification. Nous proposons un protocole qui permet de supprimer un sous-ensemble des modifications stables. Lorsque des modifications sont supprimées du journal nous disons que le journal est *tronqué*. La figure 4.2 donne un exemple d'un journal et d'une version tronquée de ce journal.

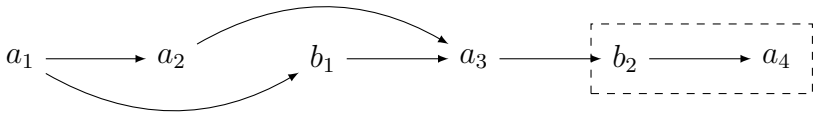


FIGURE 4.2 – Exemple du journal d'un pair. L'arrangement horizontal des opérations traduit leur ordre d'ajout dans le journal. Ainsi les opérations ont été ajoutées dans l'ordre suivant : a_1, a_2, b_1, a_3, b_2, a_4. Un arc dirigé entre deux modifications se traduit par « est une dépendance déclarée de ». La partie encadrée correspond à une version tronquée de ce journal.

Transmission d'un état authentifiable. Au lieu de récupérer un journal dans son intégralité, un pair qui rejoint une collaboration pourrait récupérer l'état d'une copie du contenu partagé. Un pair malintentionné peut construire un état de toutes pièces avant de le transmettre à un pair qui rejoint la collaboration. Le nouveau pair n'a aucun moyen de décider si l'état qu'il reçoit est authentique ou falsifié. Pour s'assurer de l'authenticité d'un état, nous proposons un protocole qui permet d'authentifier un état à partir d'un journal tronqué. Lorsqu'un pair rejoint la collaboration, il récupère un état ainsi que le journal tronqué qui lui est associé. La figure 4.3 illustre cette approche.

Nous développons d'abord le concept de stabilité dans la section 4.1. Dans la section 4.2, nous présentons un protocole qui protége la convergence des copies à l'aide de journaux intégrales. Nous nous basons sur ce protocole pour présenter dans la section 4.3 un protocole qui permet de tronquer un journal et d'authentifier un état à partir d'un journal tronqué.

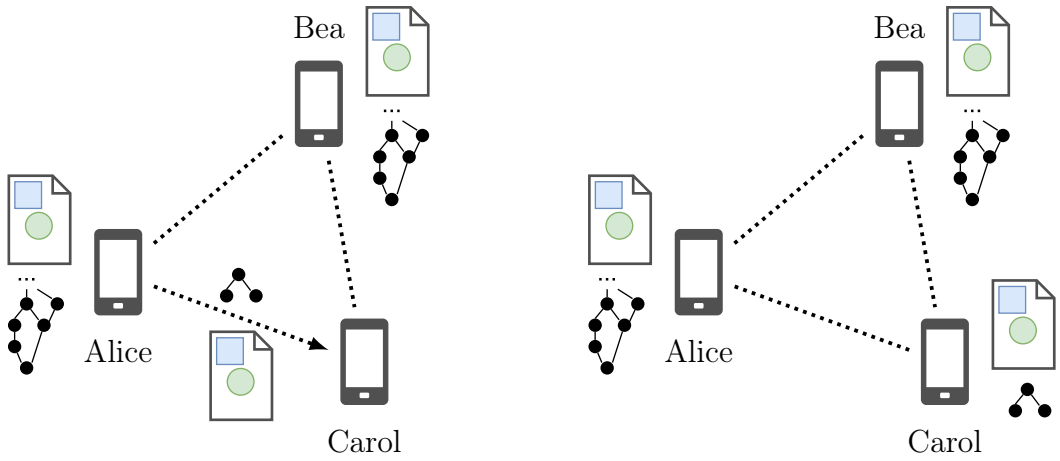


FIGURE 4.3 – Alice et Bea répliquent de manière optimiste un dessin. Elles utilisent un journal infalsifiable pour assurer la convergence des copies des pairs honnêtes. Carol veut rejoindre la collaboration. (a) Alice envoie l'état actuel de sa copie du dessin, ainsi qu'une version tronquée de son journal infalsifiable. (b) Carol vérifie l'authenticité du journal tronqué. Elle vérifie également l'authenticité de l'état reçu à partir du journal tronqué.

4.1 Stabilité

4.1.1 Généralités

Dans la littérature [13, 28, 71], la stabilité désigne souvent un prédicat qui une fois vrai le reste pour toujours. Dans ce manuscrit, nous disons qu'une modification est stable au sein d'un journal lorsqu'elle devient une dépendance de toutes les modifications qui sont ultérieurement acceptées dans le journal. Pour permettre la stabilisation de modifications, les pairs doivent accepter les modifications dans leur journal sous certaines conditions. Par exemple, si toute modification est acceptée au sein d'un journal seulement si elle dépend de toutes les modifications déjà présentes dans le journal, alors toutes les modifications enregistrées dans le journal sont stables. Les relations de dépendances forment un ordre total. Notre objectif est de trouver des contraintes d'acceptation qui permettent à terme la stabilisation de toute modification et qui sont adaptées à la réplication optimiste et à la présence de pairs malintentionnés.

Au lieu de circonscrire nos contributions à des journaux, nous décrivons ces contraintes et le concept de stabilité dans un modèle plus général. Pour éviter l'introduction d'un nouveau modèle, nous réutilisons le concept d'exécution abstraite défini dans le chapitre 3. Une exécution abstraite est suffisamment générale pour représenter une exécution ou un journal. Lorsqu'une exécution abstraite représente un journal, nous parlons d'exécution abstraite associée à un journal. Nous détaillons dans la section 4.2 comment obtenir à partir d'un journal son exécution abstraite associée. Pour simplifier la compréhension, nous utilisons dans les exemples des exécutions abstraites qui ne sont pas forcément associées à des journaux. Les contraintes d'acceptation d'une modification au sein d'un journal se traduisent par des propriétés de cohérence sur les opérations au sein d'une exécution abstraite.

Pour définir la stabilité au sein d'une exécution abstraite, la Definition 4.1 introduit le concept d'extension d'une exécution abstraite. Cette définition repose sur le concept d'extension d'une histoire introduit dans la sous-section 3.1.3 du chapitre 3.

Definition 4.1 (Extension d'une exécution abstraite)**.** Une exécution abstraite $A' \stackrel{\text{def}}{=} \langle H', \mathsf{vis}' \rangle$ est une extension d'une exécution abstraite (bien-formée ou mal-formée) $A \stackrel{\text{def}}{=} \langle H, \mathsf{vis} \rangle$, et nous écrivons $A' \sqsupset A$, si et seulement si :

— L'histoire H' de A' est une extension de l'histoire H de A :

$$H' \sqsupset H$$

— Les relations de visibilité de A sont conservées.

$$x, y \in A \wedge x \xrightarrow{\mathsf{vis}}_{A'} y \iff x \xrightarrow{\mathsf{vis}}_{A} y$$

Dans une exécution abstraite, une opération est stable une fois qu'elle est visible à toute opération future. Puisque l'acceptation d'une opération est conditionnée par le modèle de cohérence, la stabilité l'est également. Une opération peut être stable pour un modèle de cohérence donné et ne pas l'être pour un autre. La Definition 4.2 définit formellement

la stabilité. Le théorème 4.1 met en perspective le concept de stabilité et de force d'un modèle de cohérence.

Definition 4.2 (Opération stable)**.** Soit une opération x d'une exécution abstraite (bien-formée ou mal-formée) A qui respecte un modèle de cohérence C. x est C-stable dans A, et nous écrivons $\mathsf{stable}_A^C(x)$, si et seulement si il n'existe pas une opération y d'une extension A' de A telle que A' respecte C, y est absente de A, et x n'est pas visible à y.

$$\mathsf{stable}_A^C(x) \overset{\text{def}}{\iff} \forall A' \sqsupset A.\, A' \in C \implies \forall y \in A' - A.\, x \xrightarrow{\mathsf{vis}} y$$

Théorème 4.1 (Force des modèles de cohérence et stabilité)**.** *Soit deux modèles de cohérence C et C' tels que C est plus fort que C'. Si une opération d'une exécution abstraite A est C'-stable, alors elle est également C-stable.*

$$C \subset C' \wedge \mathsf{stable}_A^{C'}(x) \implies \mathsf{stable}_A^C(x)$$

Démonstration. Procédons par contradiction. Soit deux modèles de cohérence C et C' tels que C est plus fort que C'. Soit une exécution abstraite A qui respecte le modèle de cohérence C. Elle respecte donc également le modèle de cohérence C'. Supposons qu'il existe une opération x de A qui est C'-stable et n'est pas C-stable. Puisque x n'est pas C-stable, il existe donc une opération y d'une extension A' de A telle que A' respecte C, y est absente de A, et x n'est pas visible à y. A' respecte également C'. x n'est donc pas C'-stable. Ce qui contredit l'hypothèse initiale. □

Nous souhaitons qu'au cours de l'exécution un ensemble croissant d'opérations devienne stable. En d'autres termes, toute opération devrait à terme devenir stable. La Definition 4.3 exprime cette attente sous la forme d'une propriété de vivacité.

Definition 4.3 (Opération stabilisable)**.** Soit une opération x d'une exécution abstraite A qui respecte un modèle de cohérence C. x est C-stabilisable dans A si et seulement si il existe une extension de A dans laquelle x est C-stable. A garantit la propriété de vivacité $\mathsf{Stabilisable}_C$ si et seulement si toute opération de A est C-stabilisable. Un modèle de cohérence C est stabilisable, et nous écrivons $C \in \mathsf{Stabilisable}$, si et seulement si toutes les opérations de toutes les exécutions abstraites correctes et finies de C sont C-stabilisables.

$$A \in \mathsf{Stabilisable}_C \overset{\text{def}}{\iff} \forall x \in A\, \exists A' \sqsupset A.\, A' \in C \wedge \mathsf{stable}_{A'}^C(x)$$

$$C \in \mathsf{Stabilisable} \overset{\text{def}}{\iff} \forall A \in C.\, A \textit{est finie} \implies A \in \mathsf{Stabilisable}_C$$

Nous recherchons un modèle de cohérence stabilisable qui est réaliste dans le contexte de notre travail : il doit garantir la disponibilité en lecture et en écriture des copies, ainsi que la convergence à terme des copies en présence de pairs malintentionnés. Nous proposons d'abord d'explorer la notion de stabilité dans le modèle de cohérence causale et le modèle de cohérence View-Fork-Join-Causal. Ces deux modèles sont présentés dans le chapitre 3. Nous construisons ensuite deux nouveaux modèles de cohérence qui sont stabilisables et adaptés à notre contexte.

4.1.2 Stabilité causale

Baquero et al. [13] ont introduit la stabilité causale comme un moyen de réduire la taille d'un journal. La stabilité causale permet de supprimer les méta-données des opérations qui ne sont plus nécessaires au maintien de la cohérence de la copie et du journal d'un pair. Ils remarquent que la connaissance des dépendances d'une opération n'est plus nécessaire une fois que toute opération reçue ultérieurement dépend de cette opération.

Leur définition de la stabilité causale se place du point de vue d'un pair. Dans ce manuscrit, nous avons défini la stabilité sur une exécution abstraite. Cette abstraction nous permet de raisonner autant d'un point de vue global, lorsque l'exécution abstraite explique une exécution, que du point de vue d'un pair, lorsque l'exécution abstraite est associée à un journal.

Pour définir la stabilité causale, nous devons nous demander quand est-ce qu'une opération devient nécessairement visible à toute opération future. Cette question revient à se demander quand est-ce qu'une opération ne peut plus être en concurrence avec une opération future.

Le modèle de cohérence causale contraint chaque pair à observer ses anciennes opérations. Une opération d'un pair est donc nécessairement visible à toute opération future de ce même pair. Dans la figure 4.4a, l'opération a_1 sera visible aux opérations futures de son auteur p_A. En se basant sur cette contrainte et la transitivité de la visibilité, nous déduisons qu'une fois qu'une opération quelconque est visible à l'opération d'un pair, elle l'est également à toute opération future de ce même pair. Dans la figure 4.4a, a_1 est visible à l'opération b_1. Puisque b_1 est visible à toute opération future de p_B, par transitivité a_1 l'est également. Pour qu'une opération soit causalement stable, il suffit donc que chaque pair observe cette dernière. Le théorème 4.2 définit formellement la stabilité causale.

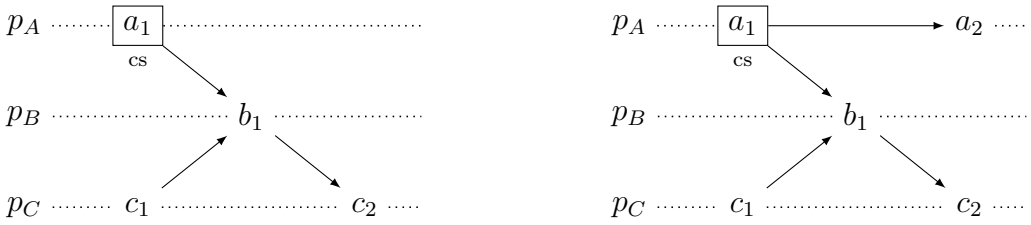


Figure 4.4 – Un ensemble de trois pairs $\mathsf{Peers} = \{p_A, p_B, p_C\}$ modifie un contenu répliqué. Dans l'exécution abstraite (a), l'opération a_1 est observée par son auteur p_A, par p_B parce que a_1 est visible à b_1, et par p_C parce que a_1 est visible à c_2. $\mathsf{obs}(a_1) = \{p_A, p_B, p_C\}$. a_1 est visible à toute opération future de p_A. b_1 et c_2 sont respectivement visibles à toute opération future de p_B et p_C. Par transitivité, a_1 est visible à toute opération future de p_B et p_C. a_1 est donc causalement stable (cs). c_1 et b_1 ne sont pas causalement stables car le pair p_A ne les observe pas. Il existe en effet une extension causale de (a), l'exécution abstraite (b), où c_1 et b_1 ne sont pas visibles à au moins une opération (a_2).

Théorème 4.2 (Stabilité causale)**.** *Dans une exécution abstraite qui respecte le modèle de cohérence causale, une opération x est causalement stable si et seulement si elle est observée par chaque pair.*

$$\mathsf{stable}_A^{\mathsf{Causal}}(x) \iff \mathsf{Peers} = \mathsf{obs}_A(x)$$

Nous introduisons la Proposition 4.1 et le Corollaire 4.2.1 pour nous aider dans la démonstration de ce théorème.

Proposition 4.1. *Soit une extension A' d'une exécution abstraite A telle que A et A' respectent les propriétés C1 (visibilité transitive) et C2 (visibilité non-spéculative). Une opération incluse dans A' et absente de A ne peut pas être visible à une opération de A.*

$$A, A' \in (C1 \cup C2) \wedge A' \sqsupset A \wedge y \in A' - A \wedge x \in A \implies y \overset{\mathsf{vis}}{\not\longrightarrow} x$$

Démonstration. Procédons par déduction. Soit x une opération de A et y une opération de A' absente de A. A' est une extension de A. Par conséquent, y ne retourne pas avant x. A et A' respectent les propriétés C1 et C2. D'après C2, si une opération ne retourne pas avant une seconde opération, alors cette dernière ne peut pas être visible à la première. y ne peut donc pas être visible à x. □

Corollaire 4.2.1. *Soit une extension A' d'une exécution abstraite A telle que A et A' respectent les propriétés C1 et C2. Soit une opération x de A et une opération y de A' absente de A. Si x n'est pas visible à y, alors x est concurrent à y.*

$$A, A' \in (C1 \cup C2) \wedge A' \sqsupset A \wedge y \in A' - A \wedge x \in A \implies x \overset{\mathsf{vis}}{\not\longrightarrow} y \implies x \| y$$

Démonstration du théorème 4.2. Le théorème est décomposable en deux implications que nous démontrons par contradiction. Soit une opération x d'une exécution abstraite A qui respecte le modèle de cohérence causale.

Démontrons $\mathsf{Peers} = \mathsf{obs}_A(x) \implies \mathsf{stable}_A^{\mathsf{Causal}}(x)$. Supposons que x n'est pas causalement stable. Il existe donc une extension causale A' de A ($A' \sqsupset A \wedge A' \in \mathsf{Causal}$) dans laquelle x n'est pas visible à une opération z de A' absente de A ($z \in A' - A \wedge x \overset{\mathsf{vis}}{\not\longrightarrow} z$). D'après le Corollaire 4.2.1, x est concurrent à z. Nous désignons par p le pair qui a exécuté z. Nous distinguons deux cas : (i) p a exécuté x (ii) p n'a pas exécuté x. Dans le premier cas, la propriété C3 est violée. A' ne respecte pas le modèle de cohérence causale. Ce qui contredit une hypothèse initiale. Dans le second cas, x est visible à au moins une opération y de A exécutée par p. Puisque A' respecte la propriété C3 du modèle de cohérence causale, y et z sont liés par la relation de visibilité. D'après la Proposition 4.1, y est visible à z. Par transitivité x est visible à z. Ce qui contredit une déduction précédente.

Démontrons $\mathsf{stable}_A^{\mathsf{Causal}}(x) \implies \mathsf{Peers} = \mathsf{obs}_A(x)$. Supposons qu'il existe une opération causalement stable x qui n'a pas été observée par un pair p ($p \notin \mathsf{obs}_A(x)$). Les éventuelles opérations de p sont visibles ou sont concurrentes à x. D'après le modèle de cohérence causale, le pair p est seulement tenu de produire une visibilité linéaire des opérations qu'il exécute. Seules ses opérations passées, ainsi que les opérations qui leurs sont visibles sont tenues d'être visibles à sa prochaine opération. Ces dernières n'incluent pas l'opération x. Il peut donc exister une opération y de p telle que x n'est pas visible à y. Ce qui rentre en contradiction avec une hypothèse initiale. □

Théorème 4.3 (Modèle de cohérence causale stabilisable)**.** *Le modèle de cohérence causale est stabilisable.*

Démonstration. Le modèle de cohérence causale ne présente pas de restrictions sur l'observation des opérations par les pairs. Toute opération est toujours observable par tout pair. Pour chaque opération, il existe donc une exécution abstraite dans laquelle l'opération est stable. Le modèle de cohérence causale est donc stabilisable. □

Le modèle de cohérence causale est stabilisable, mais il ne tolère pas la présence de pairs malintentionnés. Il n'est donc pas adapté à notre contexte. Dans la section suivante nous nous intéressons à la stabilité au sein d'un modèle de cohérence qui autorise la présence de pairs malintentionnés.

4.1.3 Stabilité View-Fork-Join-Causal

Le modèle de cohérence View-Fork-Join-Causal (VFJC) offre des garanties de convergence et de disponibilité en présence de pairs malintentionnés [47]. Pour ce faire, il relâche une contrainte de cohérence et autorise ainsi la présence d'opérations non-linéaires dans une exécution abstraite. Une opération est non-linéaire si une autre opération lui est concurrente et qu'elles sont toutes deux exécutées par le même pair. Dans la figure 4.5, m_1 et m_2 sont des opérations non-linéaires. Seuls les pairs malintentionnés peuvent être les auteurs d'opérations non-linéaires. Les opérations non-linéaires forment des embranchements.

La formation d'embranchements rend plus difficile la définition de la stabilité. Il ne suffit plus qu'un pair observe une opération x pour être assuré que cette dernière est visible à toutes ses opérations futures. En effet, le pair peut être malintentionné : il peut exécuter une opération non-linéaire et concurrente à x. Dans l'exécution abstraite A_1 de la figure 4.5, le pair malintentionné p_M observe l'opération a_1. Pour autant, dans l'extension de A_1, p_M exécute une opération m_2 concurrente à a_1.

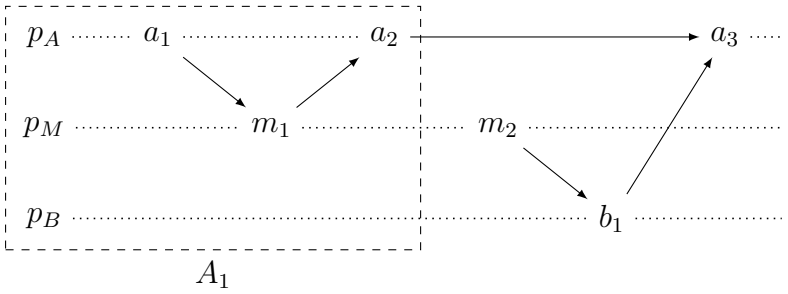


FIGURE 4.5 – Exécution abstraite qui explique la modification d'un contenu partagé par un pair malintentionné p_M, ainsi que deux pairs honnêtes p_A et p_B.

Cependant, le modèle de cohérence View-Fork-Join-Causal contraint l'acceptation d'opérations non-linéaires. En effet, les pairs ne peuvent pas accepter directement les opérations qu'ils reconnaissent comme non-linéaires. En d'autres termes, une opération y ne peut pas être immédiatement visible à une opération z ($y \downarrow z$) si x est une opération non-linéaire avec x ($x \| y$) et qu'elle est visible à z ($x \xrightarrow{\mathsf{vis}} z$). Le seul moyen qu'ils ont d'accepter une opération non-linéaire est de l'accepter indirectement par l'acceptation d'une

autre opération. Dans la figure 4.5, p_A reconnaît m_2 comme une opération non-linéaire. Il accepte indirectement m_2 par l'acceptation de l'opération b_1. Le pair p_B ne reconnaît pas encore m_2 comme non-linéaire, il peut donc l'accepter directement.

Cette contrainte a des implications intéressantes qui nous permettent de définir la stabilité. Avant de nous intéresser à ses implications générales en présence de plusieurs pairs malintentionnés, nous montrons ses implications en présence d'un seul pair malintentionné.

En présence d'un pair malintentionné p_M, une opération x devient stable lorsque les pairs honnêtes observent x et qu'ils ne peuvent plus accepter d'opérations de p_M qui sont concurrentes à x. Un pair honnête ne peut pas accepter directement une opération concurrente à x de p_M si et seulement si il a reconnu le pair comme malintentionné ou il reconnaît l'opération comme non-linéaire. D'après la Proposition 4.2, si x est visible ou égale à une opération y de p_M, alors toute opération future de p_M qui est concurrente à x est également concurrente à y. Cette opération future est non-linéaire à y. Si un pair honnête reconnaît p_M comme malintentionné ou observe une opération y de p_M telle que x lui est visible, alors il ne peut pas accepter directement une opération future de p_M qui est concurrente à x. Pour que x devienne stable il suffit donc que chaque pair honnête observe une opération de p_M qui lui est visible.

Dans la figure 4.6a, les pairs honnêtes p_A et p_B observent l'opération m_1 de p_M. Ils ne peuvent plus accepter directement d'opérations non-linéaires à m_1. m_1 et toute opération qui lui est visible (a_1) sont donc stables. Dans la figure 4.6b, le pair honnête p_A observe l'opération m_1 de p_M. Il ne peut plus accepter directement d'opérations non-linéaires à m_1. Le pair honnête p_B observe l'opération m_2 de p_M. Il ne peut plus accepter directement d'opérations non-linéaires à m_2. Toute opération visible à la fois à m_1 et m_2 est donc stable. C'est le cas de l'opération a_1. Dans la figure 4.6c, le pair honnête p_A reconnaît p_M comme un pair malintentionné. Il ne peut plus accepter directement une opération de p_M. Le pair honnête p_B observe l'opération m_2 de p_M. Il ne peut plus accepter directement d'opérations non-linéaires à m_2. a_1 et m_1 sont donc stables.

Proposition 4.2. *Soit une extension A' d'une exécution abstraite A telle que A et A' respectent le modèle de cohérence* FJC. *Soit deux opérations x et y de A et une opération z de A' absente de A. Si x est visible ou égale à y et x est concurrente à z ($x \leq_{\mathsf{vis}} y \wedge x \| z$), alors y est concurrente à z ($y \| z$).*

$$A, A' \in \mathsf{FJC} \wedge A' \sqsupset A \wedge x, y \in A \wedge x \leq_{\mathsf{vis}} y \wedge z \in A' - A \wedge x \| z \implies y \| z$$

Démonstration. Procédons par déduction. A' est une extension de A. Par conséquent, x et y retourne-avant z. D'après C1, y est visible ou est concurrent à z. Puisque x est visible ou égale à y et x est concurrente à z, y ne peut pas être visible à z. Autrement x deviendrait visible à z. y et z sont donc des opérations concurrentes. □

En présence de plusieurs pairs malintentionnés. Dans une exécution abstraite, l'opération d'un pair malintentionné est nécessairement visible à au moins une opération d'un pair honnête. Plus généralement, une chaîne de visibilité immédiate $z_1 \downarrow \cdots$ composée exclusivement d'opérations de pairs malintentionnés est visible à au moins une opération

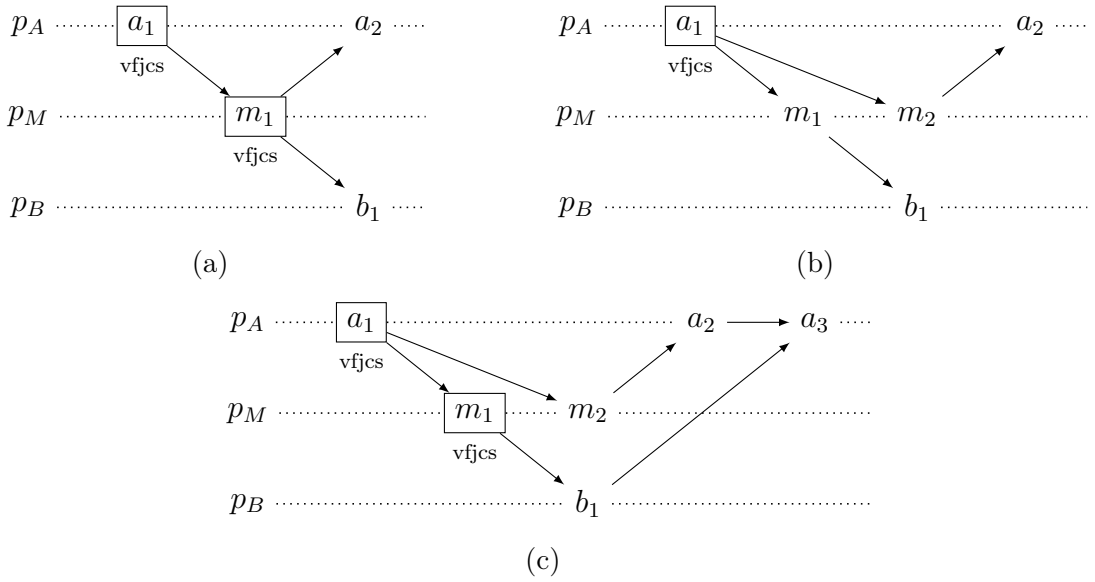


FIGURE 4.6 – Exemple d'exécutions abstraites avec des opérations vfjc-stables (vfjcs) en présence d'un seul pair malintentionné p_M et de deux pairs honnêtes p_A et p_B.

d'un pair honnête. Une opération x devient stable lorsque les pairs honnêtes (i) observent x et (ii) ne peuvent plus accepter directement une chaîne de visibilité immédiate $z_1 \downarrow \cdots \downarrow z_m$ composée exclusivement d'opérations de pairs malintentionnés dont la première opération z_1 est concurrente à x. Un pair honnête ne peut plus accepter directement une telle chaîne s'il reconnaît la dernière opération z_m de la chaîne comme non-linéaire. Nous devons donc trouver les conditions qui conduisent les pairs honnêtes à reconnaître z_m comme non-linéaire.

Afin de décomposer le problème, nous divisons l'ensemble de ces chaînes en sous-ensembles. Nous déterminons ensuite les conditions à remplir pour qu'un pair honnête rejette les châines d'un sous-ensemble donné. Pour ce faire, la Definition 4.4 introduit le concept de *chaîne d'observations cumulées*. Chaque sous-ensemble est associé à une énumération de pairs malintentionnés $P \stackrel{\text{def}}{=} \langle p_1, \ldots, p_n \rangle$ et est composé des chaînes d'observations cumulées de P. La figure 4.7 illustre l'ensemble des chaînes d'observations cumulées de l'énumération de pairs $\langle p_G, p_M, p_O \rangle$. Chaque opération de la chaîne entre g_1 et m_1 a pour auteur p_G. Chaque opération de la chaîne entre m_1 et o_1 a pour auteur p_G ou p_M. Chaque opération de la chaîne après o_1 a pour auteur p_G, p_M, ou p_O.

Definition 4.4 (Chaîne d'observations cumulées)**.** Soit une exécution abstraite A qui respecte le modèle de cohérence FJC. Une chaîne de visibilité immédiate $Z \stackrel{\text{def}}{=} z_1 \downarrow \cdots$ de A est une chaîne d'observations cumulées d'une énumération de pairs distincts $\langle p_1, \ldots, p_n \rangle$ si et seulement si elle inclut une chaîne de visibilité $Y \stackrel{\text{def}}{=} y_1 \xrightarrow{\mathsf{vis}} \cdots \xrightarrow{\mathsf{vis}} y_n$ telle que l'auteur de la i-ième opération de Y est le i-ième pair de l'énumération, et l'auteur de toute opération de Z est l'auteur d'une opération de Y qui lui est visible ou égale. On a $y_1 = z_1$.

$$\forall i \in 1\,..\,n.\, p_i = \mathsf{peer}(y_i) \wedge \forall z \in Z\, \exists y \in Y.\, y \leq_{\mathsf{vis}} z \wedge \mathsf{peer}(y) = \mathsf{peer}(z)$$

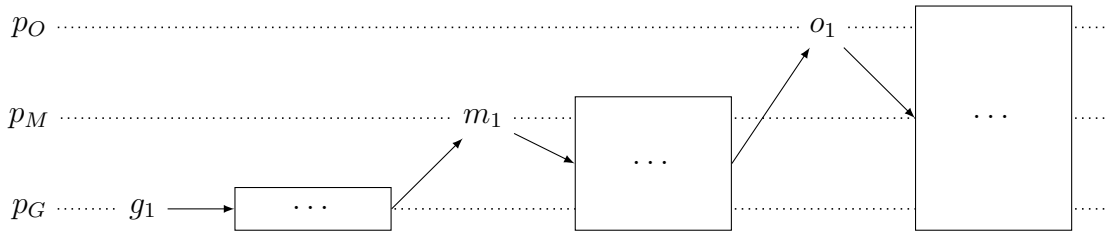


FIGURE 4.7 – Ensemble des chaînes d'observations cumulées de l'énumération de pairs $\langle p_G, p_M, p_O \rangle$.

La Proposition 4.3 présente les conditions à remplir pour qu'un pair honnête rejette les chaînes d'observations cumulées d'une énumération de pairs malintentionnés dont la première opération est concurrente à une opération x. À partir de cette proposition, nous déduisons que dans l'exécution abstraite de la figure 4.8a, l'existence de la chaîne de visibilité $g_1 \leq_{\mathsf{vis}} m_2 \leq_{\mathsf{vis}} a_2$ avec $a_1 \leq_{\mathsf{vis}} g_1$ empêche p_A d'accepter directement les chaînes d'observations cumulées de l'énumération de pairs $\langle p_G, p_M \rangle$ dont la première opération est concurrentes à a_1.

Proposition 4.3. *Soit une exécution abstraite A et une extension A' de A telles que A et A' respectent le modèle de cohérence VFJC. Soit une opération x_0 de A. Soit une énumération de pairs $P \stackrel{\text{def}}{=} \langle p_1, \ldots, p_n \rangle$ telle que $p_1, \ldots, p_{n-1}$ sont des pairs malintentionnés, et p_n est un pair honnête. A' ne peut pas contenir une chaîne d'observations cumulées Z de P dont la première opération est concurrente à x_0 si et seulement si A contient une chaîne de visibilité $X \stackrel{\text{def}}{=} x_1 \leq_{\mathsf{vis}} \cdots \leq_{\mathsf{vis}} x_n$ telle que $x_0 \leq_{\mathsf{vis}} x_1$ et le i-ième pair de P est lié à la i-ième opération de X par l'une des relations suivantes : (i) le pair est l'auteur de l'opération $p_i = \mathsf{peer}(x_i)$, ou (ii) le pair est reconnu malintentionné en l'opération $p_i \in \mathsf{knownMalicious}_A(x_i)$.*

Démonstration. $Z \stackrel{\text{def}}{=} z_1 \downarrow \cdots$ inclut une chaîne $y_1 \xrightarrow{\mathsf{vis}} \cdots \xrightarrow{\mathsf{vis}} y_n$ telle que y_i soit la première opération du pair p_i dans Z avec $1 \leq i \leq n$. On a donc $y_1 = z_1$.

Démontrons que X est suffisante pour rejeter Z. Procédons par contradiction et récurrence. Supposons que Z existe.

Montrons que pour tout j et k avec $1 \leq j < k \leq n$, si y_j est non-linéaire à x_j ou si l'auteur de y_j est reconnu malintentionné en x_j, alors y_k est concurrente à x_k. Soit z l'opération de Z qui est immédiatement visible à y_k ($z \downarrow y_k$). Par définition, l'auteur de z est l'auteur d'une opération y_l avec avec $l < k$ (éventuellement z). y_l est visible à y_k ($y_l \xrightarrow{\mathsf{vis}} y_k$). D'après notre hypothèse de récurrence, l'auteur de y_l est reconnu malintentionné en x_l ($p_l \in \mathsf{knownMalicious}_A(x_l)$) ou y_l est non-linéaire à x_l ($y_l \| x_l \wedge \mathsf{peer}(x_l) = p_l$). x_l ne peut pas être visible à z_k autrement VFJC2 serait violée. D'après le Corollaire 4.2.1, x_l et z_k sont concurrentes. Or x_l est visible ou égale à x_k ($x_l \leq_{\mathsf{vis}} x_k$). D'après la Proposition 4.2, x_k et y_k sont donc concurrentes ($x_k \| y_k$). L'auteur de y_k est soit reconnu malintentionné en x_k ou est l'auteur de x_k. Dans le dernier cas, z_k est non-linéaire à x_k.

x_0 est concurrente à z_1. D'après la Proposition 4.2, z_1 est concurrente à x_1 ($z_1 \| x_1$). Nous déduisons par récurrence que y_n est non-linéaire à x_n. p_n est donc un pair malintentionné. Ce qui contredit une hypothèse initiale.

Démontrons que X est nécessaire pour rejeter Z. Pour que y_n ne puisse pas exister, il est nécessaire qu'une opération x_n de A dont l'auteur est p_n existe et que x_n ne puisse pas être visible à y_n. D'après le Corollaire 4.2.1, si x_n n'est pas visible à y_n, alors x_n et y_n sont forcément concurrentes (et donc non-linéaires dans notre cas).

Montrons que si nous souhaitons que y_k $(k > 1)$ soit concurrente à une opération x_k de A, alors pour tout $0 < j < k$, il doit exister une opération x_j de A visible ou égale à x_k telle que x_j soit non-linéaire à y_j ou l'auteur de y_j (p_j) soit reconnu malintentionné en x_j. Soit z l'opération de Z immédiatement visible à y_k $(z \downarrow y_k)$. x_k ne peut pas être visible à y_k si et seulement si il existe une opération x_l de X visible ou égale à x_k telle que l'auteur de z est reconnu malintentionné en x_l ou z est non-linéaire à x_l. Si x_l est non-linéaire à z, alors elle est également non-linéaire à y_l. Par construction, l'auteur de z est l'un des pairs de l'énumération $p_1, \ldots, p_{k-1}$. Pour que y_k soit non-linéaire à une opération x_k de A, il doit donc exister pour tout $0 < j < k$, une opération x_j de X visible à x_k telle que x_j soit non-linéaire à y_j ou l'auteur de y_j soit reconnu malintentionné en x_j. Pour que x_1 soit non-linéaire à y_1 il est nécessaire que x_0 soit visible ou égale à x_1.

Par récurrence, nous générons un ensemble de châines de visibilité dont l'existence est nécessaire pour rejeter Z. Cet ensemble inclut la chaîne X. Pour s'en convaincre, il suffit de supposer que l'opération de Z qui est immédiatement visible à y_n a pour auteur p_{n-1} et de prendre $j = k - 1$ à chaque étape de récurrence. L'existence de X est donc nécessaire et suffisante pour rejeter Z. □

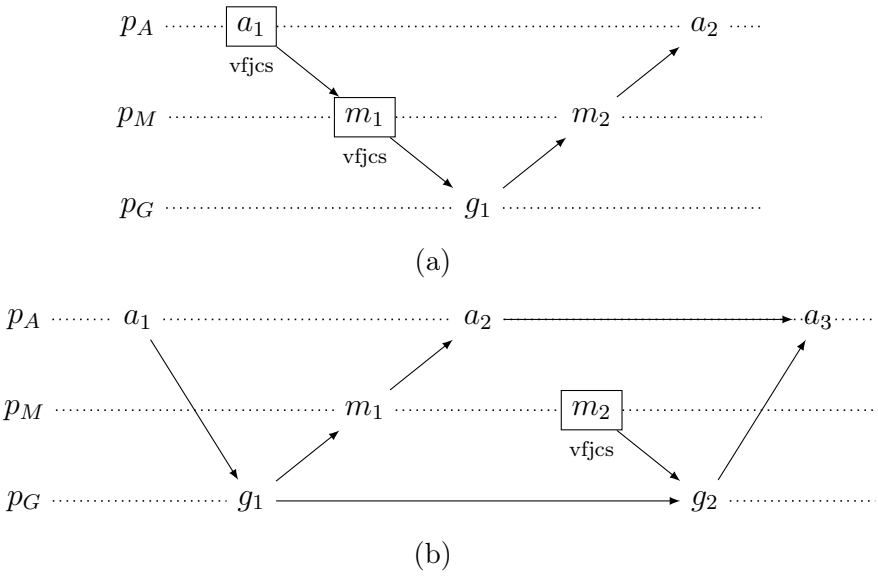


FIGURE 4.8 – Exécutions abstraites avec des opérations vfjc-stables (vfjcs) en présence d'un pair honnête p_A, et de deux pairs malintentionnés p_M et p_G.

Le rejet des chaînes d'observations cumulées d'une énumération P de pairs implique le rejet des chaînes d'observations cumulées des énumérations de pairs qui préfixent P. Par exemple, le rejet des chaînes d'observations cumulées de l'énumération $\langle p_G, p_M, p_O \rangle$, implique le rejet des chaînes d'observations cumulées des énumérations $\langle p_G, p_M \rangle$ et $\langle p_G \rangle$.

Il suffit donc de rejeter les chaînes d'observations cumulées de chaque permutation de l'ensemble des pairs malintentionnés pour les rejeter toutes. Pour un ensemble de deux pairs malintentionnés p_G et p_M, il suffit de rejeter les chaînes d'observations cumulées de $\langle p_G, p_M\rangle$ et $\langle p_M, p_G\rangle$. À partir de cette observation et de la Proposition 4.3, nous déduisons le théorème 4.4.

Prenons pour exemple les exécutions abstraites de la figure 4.8. L'ensemble des pairs de ces exécutions abstraites est composé du pair honnête p_A ($\mathsf{Honest} \stackrel{\text{def}}{=} \{p_A\}$) et des pairs malintentionnés p_G et p_M ($\mathsf{Malicious} \stackrel{\text{def}}{=} \{p_G, p_M\}$). Le produit cartésien des permutations des pairs malintentionnés par les pairs honnêtes correspond à l'ensemble d'énumérations $\mathsf{Perm}(\mathsf{Malicious}) \times \mathsf{Honest} = \{\langle p_G, p_M, p_A\rangle, \langle p_M, p_G, p_A\rangle\}$. Une opération est vfjc-stable si et seulement si pour chaque énumérations de pairs elle est visible ou égale à la première opération d'une chaîne de visibilité liée à l'énumération considérée. Dans l'exécution abstraite de la figure 4.8a, les chaînes de visibilité $g_1 \leq_{\mathsf{vis}} m_1 \leq_{\mathsf{vis}} a_2$ et $m_1 \leq_{\mathsf{vis}} g_1 \leq_{\mathsf{vis}} a_2$ sont respectivement liées aux énumérations $\langle p_G, p_M, p_A\rangle$ et $\langle p_M, p_G, p_A\rangle$. a_1 et m_1 sont visibles ou égales à la première opération de ces deux chaînes. Elles sont donc toutes deux vfjc-stables. Dans l'exécution abstraite de la figure 4.8b, les chaînes de visibilité $g_2 \leq_{\mathsf{vis}} a_3 \leq_{\mathsf{vis}} a_3$ et $m_2 \leq_{\mathsf{vis}} g_2 \leq_{\mathsf{vis}} a_3$ sont respectivement liées aux énumérations $\langle p_G, p_M, p_A\rangle$ et $\langle p_M, p_G, p_A\rangle$. m_2 est visible ou égale à la première opération de ces deux chaînes. m_2 est donc vfjc-stable.

Théorème 4.4 (Stabilité VFJC)**.** *Dans une exécution abstraite qui respecte le modèle de cohérence* VFJC*, une opération x_0 est vfjc-stable si et seulement si les pairs honnêtes observent x_0 et pour chaque énumération de pairs $\langle p_1, \ldots, p_n\rangle$ issue du produit cartésien des permutations des pairs malintentionnés par les pairs honnêtes, il existe une chaîne de visibilité $x_1 \leq_{\mathsf{vis}} \cdots \leq_{\mathsf{vis}} x_n$ telle que $x_0 \leq_{\mathsf{vis}} x_1$ et le i-ième pair de l'énumération est lié à la i-ième opération de l'énumération par l'une des relations suivantes : (i) le pair est l'auteur de l'opération $p_i = \mathsf{peer}(x_i)$, ou (ii) le pair est reconnu malintentionné en l'opération $p_i \in \mathsf{knownMalicious}_A(x_i)$.*

$$
\begin{aligned}
&\mathsf{stable}_A^{\mathsf{VFJC}}(x_0) \iff \\
&\quad \mathsf{Honest} \subseteq \mathsf{obs}_A(x_0) \wedge \\
&\quad \forall \langle p_1, \ldots, p_n\rangle \in \mathsf{Perm}(\mathsf{Malicious}) \times \mathsf{Honest}\ \exists x_0 \leq_{\mathsf{vis}} \cdots \leq_{\mathsf{vis}} x_n \\
&\quad \forall i \in 1\,..\,n.\, (p_i = \mathsf{peer}(x_i) \vee p_i \in \mathsf{knownMalicious}_A(x_i))
\end{aligned}
$$

Démonstration. En l'absence de pairs malintentionnés, la démonstration est identique à celle du théorème 4.2. En présence d'au moins un pair malintentionné, une opération x_0 devient stable si et seulement si les pairs honnêtes observent x_0 et ne peuvent plus accepter de chaîne d'observations cumulées d'une énumération de pairs malintentionnés dont la première opération est concurrente à x_0. À partir de la Proposition 4.3, nous déduisons qu'une opération x_0 devient stable si et seulement si pour chaque permutation de l'ensemble des pairs malintentionnés et pour chaque pair honnête, le pair honnête observe une chaîne de visibilité liée à la permutation de pairs telle que x_0 est visible ou égale à la première opération de cette chaîne. Dans le théorème, l'observation du pair honnête est matérialisée dans la chaîne sous la forme d'une opération. C'est pourquoi nous considérons des chaînes liées aux énumérations issues du produit cartésien de l'ensemble des permutations de l'ensemble des pairs malintentionnés par l'ensemble des pairs honnêtes. □

Théorème 4.5 (Modèle de cohérence VFJC stabilisable)**.** *Le modèle de cohérence VFJC est stabilisable.*

Démonstration. Le modèle de cohérence VFJC borne le nombre d'embranchements que les pairs honnêtes peuvent accepter [47]. Un pair malintentionné ne peut donc pas exécuter une infinité d'opération concurrente à toute opération x. Le modèle de cohérence VFJC permet la jonction des embranchements. Dès lors qu'un pair honnête observe une opération, tout pair honnête peut l'observer au moins de manière indirect. Toute opération x est donc à terme visible à toute opération future. □

Le modèle de cohérence VFJC est stabilisable (théorème 4.5) et tolère la présence de pairs malintentionnés. Cependant, la stabilisation d'une opération requiert le concours de l'ensemble des pairs qui ne sont pas évincés. Dans les deux sous-sections suivantes nous définissons des modèles de cohérence qui prennent en compte la nature dynamique des groupes de pairs.

4.1.4 Stabilité causale dynamique

Dans les sections précédentes, nous avons montré qu'une opération, au sein d'une exécution abstraite qui respecte le modèle de cohérence causale ou le modèle de cohérence VFJC peut à terme devenir stable. Dans le cas d'une exécution abstraite qui respecte le modèle de cohérence causale, tous les pairs doivent observer une opération pour que cette dernière soit stable. Dans le cas d'une exécution abstraite qui respecte le modèle de cohérence VFJC, la participation de tous les pairs à l'exception des pairs évincés à terme est nécessaire pour stabiliser une opération. En pratique ces conditions rendent inutilisable la stabilité à moins de restreindre le groupe de pairs à un petit nombre de pairs connus. Si nous considérons un ensemble (dénombrable) infini de pairs, un pair qui n'a pas encore exécuté une opération peut toujours exécuter une opération concurrente à toute opération déjà exécutée. Dans cette section nous définissons le modèle de cohérence causale dynamique qui invalide cette capacité et repose sur le modèle de cohérence causale. Il permet de stabiliser des opérations avec la participation d'un sous-ensemble de pairs.

Au cours du temps, les pairs joignent la collaboration. La composition du groupe de pairs varie tout au long de la collaboration : le groupe est dynamique. A un instant donné il y a donc un ensemble de pairs qui a joint la collaboration et un ensemble complémentaire de pairs qui n'a pas encore joint la collaboration. Nous introduisons une opération d'invitation pour suivre l'évolution de la composition du groupe.

Nous restreignons la capacité d'un nouveau pair à exécuter des opérations concurrentes à d'autres en introduisant une contrainte supplémentaire. Un pair hérite des observations du pair qui l'invite. Si un pair observe une opération x, alors le pair qu'il invite par la suite observe également x. x est alors visible à toute opération exécutée par le pair invité. Le pair invité ne peut pas exécuter une opération concurrente à x.

Du point de vue d'un protocole, l'héritage des observations d'un pair peut traduire la transmission du journal des modifications ou de l'état de la copie. L'état de la copie du pair invité devient équivalente à celle du pair qui l'invite. Les opérations qu'il exécute sont donc potentiellement influencées par les modifications qui ont amenées à la constitution de cet état.

Pour simplifier, nous supposons qu'une opération d'invitation invite un seul pair et qu'un pair ne peut pas être invité plusieurs fois. Cette dernière simplification permet d'éviter en particulier les invitations concurrentes qui sont la source d'une complexité supplémentaire pour définir la stabilité. L'invitation d'un pair est visible à toutes les opérations de ce pair. Seul le pair qui initie la collaboration s'invite lui-même dans sa première opération. La Definition 4.6 regroupe ces contraintes au sein du modèle de cohérence *Invitation*. La Definition 4.5 présente le modèle de cohérence causale dynamique qui correspond à la conjonction du modèle de cohérence causale et du modèle de cohérence *Invitation*.

Definition 4.5 (Cohérence causale dynamique)**.** Soit une exécution abstraite A. A respecte le modèle de cohérence causale dynamique, et nous écrivons $A \in \mathsf{DynCausal}$, si et seulement si il respecte à la fois le modèle de cohérence causale et le modèle de cohérence invitation.

$$\mathsf{DynCausal} \stackrel{\text{def}}{=} \mathsf{Causal} \cap \mathsf{Invitation}$$

Definition 4.6 (Modèle de cohérence Invitation)**.** Soit une exécution abstraite A. A respecte le modèle de cohérence Invitation, et nous écrivons $A \in \mathsf{Invitation}$, si et seulement si :

I1 (opération d'invitation) Le type de données répliquées T dispose d'une opération d'invitation invite.

$$\{\mathsf{invite}(p) \mid p \in \mathsf{Peers}\} \subseteq \mathsf{Op}_T$$

I2 (opération d'invitation initiale) Si une opération invite son auteur, alors elle est visible à toutes les autres opérations. Elle est unique.

$$x, y \in A \wedge \mathsf{call}(x) = \mathsf{invite}(\mathsf{peer}(x)) \implies x \leq_{\mathsf{vis}} y$$

I3 (invitation exigée) L'invitation d'un pair est visible ou égal à chaque opération exécutée par ce pair.

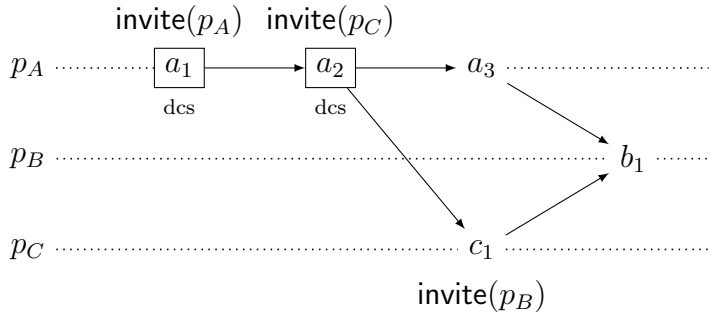
$$x, y \in A \wedge \mathsf{call}(x) = \mathsf{invite}(\mathsf{peer}(y)) \implies x \leq_{\mathsf{vis}} y$$

I4 (unicité des invitations) Un pair est invité une seule fois. Deux invitations ne peuvent pas inviter le même pair.

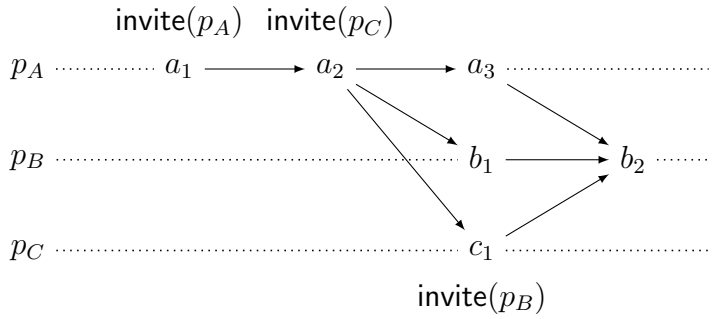
$$p, q \in \mathsf{Peers} \wedge x, y \in A \wedge \mathsf{call}(x) = \mathsf{invite}(p) \wedge \mathsf{call}(y) = \mathsf{invite}(q) \implies p \neq q$$

Le modèle de cohérence causale dynamique est plus fort que le modèle de cohérence causale. Les remarques qui ont aidées à définir la stabilité causale s'appliquent également pour la stabilité causale dynamique. Lorsqu'un pair observe une opération, il ne peut plus exécuter d'opérations concurrentes à cette dernière.

Si un pair invite un nouveau pair, alors ce dernier hérite de l'ensemble des observations du pair qui l'invite. Dans la figure 4.9a, le pair honnête invite p_B en c_1. A ce moment, p_C observe c_1, a_2, et a_1. Puisque l'invitation c_1 doit être visible aux opérations de p_B, p_B observe nécessairement c_1, a_2, et a_1.



(a)



(b)

FIGURE 4.9 – L'exécution abstraite (a) respecte le modèle de cohérence causale dynamique. Les opérations DynCausal-stables (dcs) sont encadrées. L'exécution abstraite (b) ne respecte pas le modèle de cohérence causale dynamique. La propriété I3 (invitation exigée) est violée. En effet, il n'existe pas une invitation de p_B qui est visible à b_1.

À partir de ces deux remarques nous en déduisant que seuls les pairs dont l'invitation est visible ou est concurrente à une opération x peuvent exécuter une opération concurrente à x. x devient stable dès lors que ces pairs l'observent. Nous parlons d'*observateurs requis*. La Definition 4.7 définit formellement les observateurs requis d'une opération.

Definition 4.7 (Observateurs honnêtes requis)**.** Un pair honnête est un observateur requis d'une opération x si et seulement si il est invité dans une opération qui est visible ou concurrente à x.

$$\mathsf{rqHonestObs}_A(x) \stackrel{\text{def}}{=} \{q \in \mathsf{Honest} \mid \exists y \in A.\, \mathsf{call}(y) = \mathsf{invite}(q) \wedge x \not\leq_{\mathsf{vis}} y\}$$

Nous en déduisons ainsi la stabilité causale dynamique décrite dans le théorème 4.6. Le tableau 4.1 indique les observateurs et les observateurs requis des opérations de la figure 4.9a. À partir de ces informations nous concluons que les opérations a_1 et a_2 sont stables.

Théorème 4.6 (Stabilité causale dynamique)**.** *Dans une exécution abstraite qui respecte le modèle de cohérence causale dynamique, une opération x est stable si les observateurs requis de x sont inclus dans les observateurs de x.*

$$\mathsf{stable}_A^{\mathsf{DynCausal}}(x) \iff \mathsf{rqHonestObs}_A(x) \subseteq \mathsf{obs}_A(x)$$

opération	rqHonestObs	obs
a_1	$\emptyset$	$\{p_A, p_B, p_C\}$
a_2	$\{p_A\}$	$\{p_A, p_B, p_C\}$
a_3	$\{p_A, p_B, p_C\}$	$\{p_A\}$
c_1	$\{p_A, p_C\}$	$\{p_B, p_C\}$
b_1	$\{p_A, p_B, p_C\}$	$\{p_B\}$

TABLE 4.1 – Observateurs requis et observateurs des opérations de l'exécution abstraite de la figure 4.9a.

Démonstration. L'équivalence est décomposable en deux implications. Nous prouvons la première par contradiction. La seconde est triviale et peut être simplement déduite de l'explication déroulée tout au long de cette section. Soit une exécution abstraite A qui respecte le modèle de cohérence causale dynamique. Soit une opération x de A.

Démontrons $\mathsf{rqHonestObs}_A(x) \subseteq \mathsf{obs}_A(x) \implies \mathsf{stable}_A^{\mathsf{DynCausal}}(x)$**.** x est observée par l'ensemble de ses observateurs requis. Chacun d'entre eux est invité dans une opération qui est visible ou est concurrente à x. Supposons que x n'est pas stable. Il existe donc une extension A' de A telle que A' respecte le modèle de cohérence causale dynamique et x n'est pas visible à une opération y de $A' - A$. Nous distinguons deux cas. Si l'auteur de y est un observateur requis de x, alors x est visible à une opération incluse dans A de l'auteur de y. Suivant la propriété C3 (ordre linéaire pour chaque pair), cette opération est visible à y. Par transitivité, x est visible à y. Ce qui contredit une hypothèse. Si l'auteur de y n'est pas un observateur requis, alors il est invité en x ou dans une opération z telle que x est visible à z. Suivant la propriété C3, x ou z est visible à y. Par transitivité, x est visible à y. Ce qui contredit une hypothèse. □

Théorème 4.7 (Modèle de cohérence causale dynamique stabilisable)**.** *Le modèle de cohérence causale dynamique est stabilisable.*

Démonstration. Le modèle de cohérence causale dynamique ne présente pas de restrictions sur l'observation des opérations par les pairs invités. Toute opération est toujours observable par tout pair invité. Pour chaque opération, il existe donc une exécution abstraite où l'opération est stable. Le modèle de cohérence causale dynamique est donc stabilisable. □

Pour être réaliste, nous devrions également prendre en compte les pairs qui quittent une collaboration. Par simplicité nous ne les prenons pas un compte. Les pairs pourraient quitter une collaboration à l'aide d'une opération dédiée à cet effet. Le pair serait alors évincé.

Le modèle de cohérence causale dynamique permet à des opérations de devenir stable (théorème 4.7) dans un contexte où le groupe de pairs évolue au cours de la collaboration. Similairement au modèle de cohérence causale, il ne tolère pas la présence de pairs malintentionnés. Dans la section suivante nous proposons de définir un nouveau modèle de cohérence basé sur les modèles de cohérence VFJC et *Invitation* pour définir une stabilité adaptée aux groupes dynamiques.

4.1.5 Stabilité View-Fork-Join-Causal dynamique

Le modèle de cohérence VFJC tolère la présence de pairs malintentionnés. Il inclut toutefois des contraintes qui permettent à terme d'évincer les pairs reconnus malintentionnés. Cette caractéristique nous a permis de définir la stabilité VFJC. Tous les pairs, à l'exception des pairs évincés à terme, doivent participer pour qu'une opération devienne stable. C'est une condition réaliste lorsque le groupe de pairs est de taille raisonnable et que la composition du groupe est connue à l'initialisation de la collaboration.

Le modèle de cohérence causale dynamique nous a permis de stabiliser une opération avec un sous-ensemble de l'ensemble des pairs qui peuvent joindre la collaboration. Il exige qu'un pair soit invité avant qu'il puisse exécuter des opérations. L'invitation d'un pair est rendue explicite à l'aide d'une opération qui est visible à toutes les opérations du pair. Il est ainsi possible de définir une stabilité adaptée aux groupes dynamiques. Dans cette section nous proposons un modèle de cohérence qui repose sur les modelés de cohérence VFJC et *Invitation* qui nous permet de définir une stabilité adaptée aux groupes dynamiques en présence de pairs malintentionnés.

La conjonction des modèles de cohérence VFJC et *Invitation* ne permet pas d'aboutir à une telle définition. En effet, si nous considérons un ensemble (dénombrable) infini de pairs composé d'au moins un pair malintentionné, un pair malintentionné peut toujours inviter un autre pair dans une opération concurrente à une opération x. Toute opération x ne peut donc pas devenir stable sans la participation de l'ensemble des pairs. Dans la figure 4.10, le pair malintentionné p_M invite un pair p_G dans un embranchement dont il est à l'origine. p_A reconnaît que p_M est malintentionné, mais il présume honnête le pair p_G. p_A est forcé d'accepter l'opération linéaire g_1.

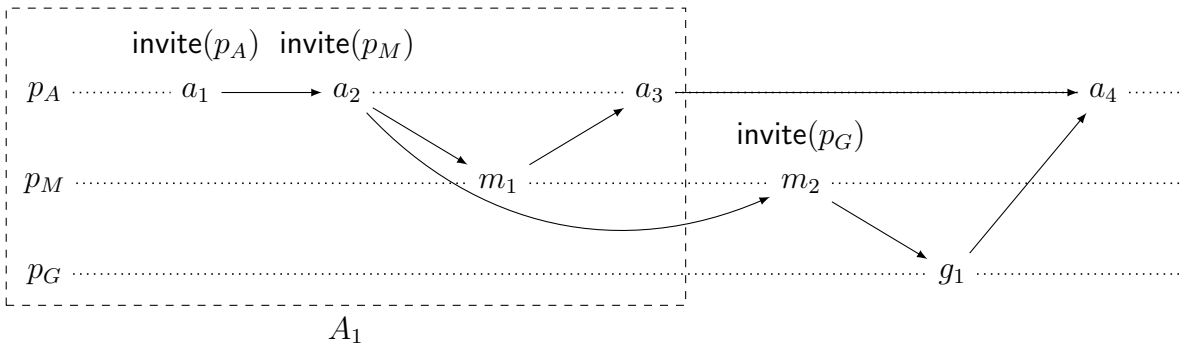


FIGURE 4.10 – Exemple d'une exécution abstraite qui respecte les modèles de cohérence *VFJC* et *Invitation* dans laquelle le pair malintentionné p_M invite le pair p_G dans un nouvel embranchement.

Pour pouvoir définir une stabilité adaptée à des groupes dynamiques, il nous faut rejeter ce type d'embranchements. Dans la figure 4.10, p_A devrait rejeter l'embranchement $m_2 \downarrow g_1$. Quel que soit le nombre d'invitations effectuées dans l'embranchement, il existe toujours une première invitation effectuée par le pair à l'origine de l'embranchement. Cette première invitation est une opération non-linéaire. Dans la figure 4.10, il s'agit de l'opération d'invitation m_2. Une manière simple de rejeter de tels embranchements est de contraindre les pairs à observer l'invitation d'un pair avant d'observer des opérations de

ce pair. Puisque l'invitation est non-linéaire, ils la rejettent et sont donc dans l'incapacité d'accepter l'embranchement. Cette contrainte pourrait être appliquée seulement lorsque des branches sont jointes. Pour ce faire, nous pourrions rejeter les relations de visibilité immédiate entre une opération x et une opération y si l'auteur de x n'est pas un *invité connu* dans les autres opérations qui sont immédiatement visibles à y. La Definition 4.8 définit formellement ce que nous nommons *invité connu*. Le tableau 4.2 indique les invités connus de chaque opération de l'exécution abstraite de la figure 4.10. Dans la figure 4.10, a_3 et g_1 sont immédiatement visibles à a_4. Le pair p_A est un invité connu en g_1. En revanche, p_G n'est pas un invité connu en a_3. La jonction de ces deux branches n'est donc pas autorisée.

Definition 4.8 (Invités connus en une opération)**.** Soit une opération y d'une exécution abstraite A qui respecte le modèle de cohérence *Invitation*. Nous disons que le pair p est un invité connu en y si et seulement si y est une invitation qui invite p ou il existe une invitation qui invite p et est visible à y. $\mathsf{invited}_A(y)$ désigne l'ensemble des pairs invités connus en y.

$$\mathsf{invited}_A(y) \stackrel{\text{def}}{=} \{p \in \mathsf{Peers} \mid \exists x \in A.\, x \leq_{\mathsf{vis}} y \wedge \mathsf{call}(x) = \mathsf{invite}(p)\}$$

opération $x \in A$	$\mathsf{invited}_A(x)$	$\mathsf{invitationObs}_A(x, p_G)$
a_1	$\{p_A\}$	$\emptyset$
a_2	$\{p_A, p_M\}$	$\emptyset$
a_3	$\{p_A, p_M\}$	$\emptyset$
a_4	$\{p_A, p_M, p_G\}$	$\{p_M, p_G\}$
m_1	$\{p_A, p_M\}$	$\emptyset$
m_2	$\{p_A, p_M, p_G\}$	$\emptyset$
g_1	$\{p_A, p_M, p_G\}$	$\{p_M\}$

TABLE 4.2 – Invités connus et observateurs connus d'une invitation de l'exécution abstraite de la figure 4.9a.

Contraindre chaque pair à observer l'invitation d'un nouveau pair avant d'observer les opérations de ce nouveau pair est un peu trop strict. En effet, des exécutions abstraites qui pourraient être stabilisables sont rejetées. En réalité, nous n'avons pas besoin que chaque pair observe l'invitation, il suffit d'un seul pair qui n'est pas reconnu malintentionné. Lorsqu'un pair joint deux branches, il s'assure ainsi que pour chaque opération qui est immédiatement visible à la jonction, l'invitation de l'auteur de l'opération est observée par un invité connu dans chaque branche jointe. Cet observateur ne doit pas être reconnu malintentionné à la jonction. La Definition 4.9 définit ce que nous nommons observateur connu d'une invitation d'un pair en une opération. Le tableau 4.2 indique les observateurs connus de l'invitation du pair p_G en chaque opération de l'exécution abstraite de la figure 4.10. Dans la figure 4.10, l'invitation de p_G est observée par p_G et p_M avant l'exécution de a_1. p_G n'est pas un invité connu en a_3. p_M est un invité connu en a_3. Cependant, il est reconnu malintentionné en a_4. La jonction n'est donc pas autorisée. Nous définissons ainsi le modèle de cohérence VFJC dynamique dans la Definition 4.10.

Definition 4.9 (Observateurs connus d'une invitation)**.** Soit une opération z d'une exécution abstraite A qui respecte le modèle de cohérence *Invitation*. Nous disons que le pair p est un observateur connu en z de l'invitation d'un pair q si et seulement si p a exécuté une opération telle que l'invitation de q lui est visible ou égale et elle est visible à z. $\mathsf{invitationObs}_A(z, q)$ désigne l'ensemble des observateurs connus en z de l'invitation de q.

$$\mathsf{invitationObs}_A(z, q) \stackrel{\text{def}}{=} \left\{ p \in \mathsf{Peers} \;\middle|\; \exists x, y \in A.\, x \leq_{\mathsf{vis}} y \xrightarrow{\mathsf{vis}} z \wedge \mathsf{call}(x) = \mathsf{invite}(q) \wedge \mathsf{peer}(y) = p \right\}$$

Definition 4.10 (Cohérence View-Fork-Join-Causal dynamique)**.** Soit une exécution abstraite A. A respecte le modèle de cohérence *DyncVFJC*, et nous écrivons $A \in \mathsf{DyncVFJC}$, si et seulement si :

DVFJC1 L'exécution abstraite A respecte les modèles de cohérence VFJC et Invitation.

$$A \in \mathsf{VFJC} \cap \mathsf{Invitation}$$

DVFJC2 (rejet des invitations non-linéaires) L'invitation de l'auteur d'une opération x qui est immédiatement visible à une opération z doit être observée par au moins un invité connu qui n'est pas reconnu malintentionné en z dans chaque opération y qui est immédiatement visible à z.

$$x \downarrow z \wedge y \downarrow z \implies \exists p \in \mathsf{invited}_A(y) - \mathsf{knownMalicious}_A(z).\, p \in \mathsf{invitationObs}_A(\mathsf{z}, \mathsf{peer}(x))$$

Lorsqu'un pair est invité dans un embranchement rejeté par les autres pairs, il est exclu de manière permanente par les autres pairs. Pour respecter notre définition initiale de la stabilité au sein d'une exécution abstraite, nous considérons un tel pair comme malintentionné. Un collaborateur ou une collaboratrice attaché·e à un tel pair pourrait détecter cette exclusion. Il ou elle pourrait donc demander à un autre pair de la collaboration de l'inviter pour obtenir une nouvelle identité (un nouveau pair).

Les raisonnements élaborés dans la sous-section 4.1.3 peuvent être en partie repris. Au lieu de considérer l'ensemble des pairs, nous nous restreignons aux pairs invités. Il suffit donc que chaque pair honnête invité observe pour chaque permutation de pairs malintentionnés invités, une chaîne de visibilité correspondante pour que les opérations qui sont visibles ou sont égales à la première opération de chacune de ces chaînes soient stables.

Cependant, il est nécessaire de prendre en compte les contraintes introduites par les invitations. L'invitation d'un pair est visible à toutes les opérations de ce pair. Si un pair est invité après une opération x, alors il ne peut pas exécuter d'opération concurrente à x. Pour faciliter l'expression de la stabilité, la Definition 4.11 introduit l'ensemble des pairs invités au sein d'une exécution abstraite et des sous-ensembles correspondant. La Proposition 4.4 indique les conditions à rencontrer pour exclure les chaînes d'observations cumulées d'une énumération de pairs malintentionnés.

Definition 4.11 (Invités)**.** Soit une exécution abstraite A qui respecte le modèle de cohérence *Invitation*. Nous notons $\mathsf{Invited}_A$, l'ensemble des pairs invités en A. $\mathsf{InvitedHonest}_A$ et $\mathsf{InvitedMalicious}_A$ sont respectivement les pairs honnêtes invités en A et les pairs malintentionnés invités en A.

$$\mathsf{Invited}_A \stackrel{\text{def}}{=} \{p \in \mathsf{Peers} \mid \exists x \in A.\, \mathsf{call}(x) = \mathsf{invite}(p)\}$$

$$\mathsf{InvitedHonest}_A \stackrel{\text{def}}{=} \mathsf{Invited}_A \cap \mathsf{Honest}$$

$$\mathsf{InvitedMalicious}_A \stackrel{\text{def}}{=} \mathsf{Invited}_A \cap \mathsf{Malicious}$$

Proposition 4.4. *Soit une exécution abstraite A et une extension A' de A telles que A et A' respectent le modèle de cohérence* DynVFJC*. Soit une opération x_0 de A. Soit une énumération de pairs $P \stackrel{\text{def}}{=} \langle p_1, \ldots, p_n \rangle$ telle que $p_1, \ldots, p_{n-1}$ sont des pairs malintentionnés, et p_n est un pair honnête. A' ne peut pas contenir une chaîne d'observations cumulées Z de P dont la première opération est concurrente à x_0 si et seulement si A contient une chaîne de visibilité $X \stackrel{\text{def}}{=} x_1 \leq_{\mathsf{vis}} \cdots \leq_{\mathsf{vis}} x_n$ telle que $x_0 \leq_{\mathsf{vis}} x_1$, pour les k premières opérations $(0 \leq k \leq n)$ le i-ième pair de l'énumération est l'auteur de la i-ième opération de la chaîne ou est reconnu malintentionné dans la i-ième opération de la chaîne, et si $k \neq n$, alors la $(k+1)$-ième opération de la chaîne invite le $(k+1)$-ième pair de l'énumération.*

Démonstration. Toutes les opérations de X sont des opérations de A. Les pairs reconnus malintentionnés dans une opération de X et les auteurs des opérations de X sont donc des pairs invités de A ($\forall x \in X.\, \mathsf{peer}(x) \in \mathsf{Invited}_A \wedge \mathsf{knownMalicious}_A(x) \subset \mathsf{Invited}_A$).

Si il ne peut exister une chaîne X avec $k \neq n$, alors la démonstration est identique à celle de la Proposition 4.3.

Démontrons pour $k \neq n$ que X est suffisante pour rejeter Z. Procédons par contradiction. Supposons que Z existe. À partir de la démonstration du Proposition 4.3, nous déduisons que pour tout $j \leq k+1$, y_j est concurrente à x_j. Or x_{k+1} invite l'auteur de y_{k+1}. D'après la propriété I3 (invitation exigée) x_{k+1} doit être visible à y_{k+1}. Ce qui contredit une hypothèse initiale. X est donc suffisante pour rejeter Z.

Démontrons pour $k \neq n$ que X est nécessaire pour rejeter Z. Puisque x_{k+1} invite l'auteur de y_{k+1}, x_{k+1} doit être visible à y_{k+1}. Si cette relation de visibilité ne peut exister sans violer une autre propriété, alors la chaîne Z ne peut exister. À partir de la démonstration du Proposition 4.3, nous déduisons que la chaîne X est nécessaire pour rejeter Z. □

Le rejet des chaînes d'observations cumulées d'une énumération P de pairs n'implique pas forcément le rejet des chaînes d'observations cumulées des énumérations de pairs qui préfixent P. En effet, si la $k+1$-ième opération invite le $k+1$-ième pair (quand $k \neq n$), alors le pair honnête p_n n'est pas tenu d'observer la chaîne $x_1 \leq_{\mathsf{vis}} \cdots \leq_{\mathsf{vis}} x_k$ qui permet l'exclusion des chaînes d'observations cumulées de l'énumération de pairs $p_1, \ldots, p_k, p_n$.

Une opération x_0 est stable si et seulement si chaque pair honnête invité observe x_0 et pour toute énumération de pairs malintentionnés invités distincts le pair honnête considéré ne peut plus accepter une chaîne d'observations cumulées de l'énumération de pairs considérée. Nous aboutissons ainsi au théorème 4.8.

Théorème 4.8 (Stabilité *DynVFJC*)**.** *Soit une exécution abstraite A qui respecte le modèle de cohérence* DynVFJC*. Une opération x_0 est stable si et seulement si les observateurs honnêtes requis observent x_0 et pour chaque énumération de pairs $\langle p_1, \ldots, p_n \rangle$ issue du produit cartésien des énumérations des pairs malintentionnés invités distincts par les pairs honnêtes invités, il existe une chaîne de visibilité $x_1 \leq_{\mathsf{vis}} \cdots \leq_{\mathsf{vis}} x_n$ telle que $x_0 \leq_{\mathsf{vis}} x_1$, pour les k premières opérations ($k \geq 0$) le i-ième pair de l'énumération est l'auteur de la i-ième opération de la chaîne ou est reconnu malintentionné dans la i-ième opération de la chaîne, et si $k \neq n$, alors x_{k+1} invite p_{k+1}.*

$$
\begin{aligned}
&\mathsf{stable}_A^{\mathsf{DynVFJC}}(x_0) \iff \\
&\quad \mathsf{rqHonestObs}_A(x_0) \subseteq \mathsf{obs}_A(x_0) \wedge \\
&\quad \forall M \subseteq \mathsf{InvitedMalicious}_A.\, M \neq \emptyset \implies \\
&\quad \forall \langle p_1, \ldots, p_n \rangle \in \mathsf{Perm}(M) \times \mathsf{InvitedHonest}_A\ \exists k \in 0\,..\,n \\
&\quad \exists x_0 \leq_{\mathsf{vis}} \cdots \leq_{\mathsf{vis}} x_n.\, (k \neq n \implies \mathsf{call}(x_{k+1}) = \mathsf{invite}(p_{k+1})) \wedge \\
&\quad \forall i \in 1\,..\,k.\, p_i = \mathsf{peer}(x_i) \vee p_i \in \mathsf{knownMalicious}_A(x_i)
\end{aligned}
$$

Démonstration. En l'absence de pairs malintentionnés, la démonstration est identique à celle du théorème 4.6.

En présence d'au moins un pair malintentionné, une opération x_0 devient stable si et seulement si les observateurs honnêtes requis de x_0 observent x_0 et ne peuvent plus accepter de chaînes d'observations cumulées d'une énumération de pairs malintentionnés dont la première opération est concurrente à x_0. À partir de la Proposition 4.4, nous déduisons qu'une opération x_0 devient stable si et seulement si pour chaque énumération de pairs malintentionnés invités distincts et pour chaque pair honnête invité, le pair honnête observe une chaîne de visibilité liée à l'énumération de pairs telle que x_0 est visible ou égale à la première opération de cette chaîne. Dans le théorème, l'observation du pair honnête est matérialisée dans la chaîne sous la forme d'une opération. C'est pourquoi nous considérons des chaînes liées aux énumérations issues du produit cartésien de l'ensemble des énumérations de pairs malintentionnés invités distincts par l'ensemble des pairs honnêtes invités. □

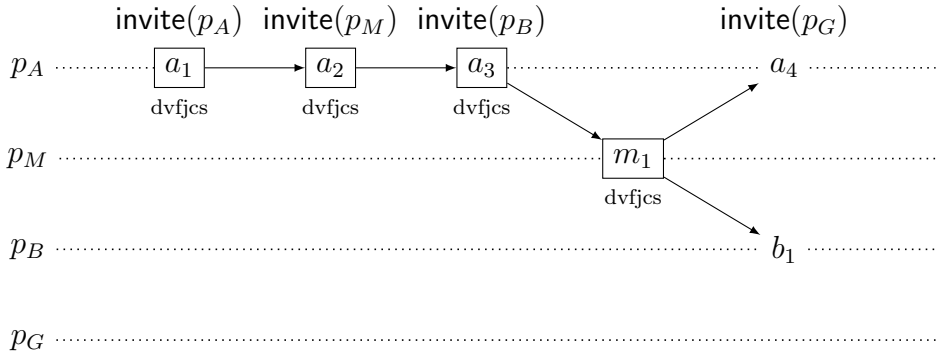


FIGURE 4.11 – Exemple d'une exécution abstraite avec des opérations *DynVFJC*-stables (dvfjjcs).

Prenons pour exemple l'exécution abstraite A de la figure 4.11. L'ensemble des pairs invités est composé de deux pairs malintentionnés p_G et p_M ($\mathsf{InvitedMalicious}_A = \{p_G, p_M\}$), ainsi que de deux pairs honnêtes p_A et p_B ($\mathsf{InvitedHonest}_A = \{p_A, p_B\}$). Le produit cartésien des énumérations des pairs malintentionnés invités distincts par les pairs honnêtes invités correspond à l'ensemble des énumérations suivantes : $\langle p_G, p_A\rangle$, $\langle p_G, p_B\rangle$, $\langle p_M, p_A\rangle$, $\langle p_M, p_B\rangle$, $\langle p_G, p_M, p_A\rangle$, $\langle p_G, p_M, p_B\rangle$, $\langle p_M, p_G, p_A\rangle$, et $\langle p_M, p_G, p_B\rangle$. Une opération est stable si et seulement si pour chaque énumération de pairs elle est visible ou égale à la première opération d'une chaîne de visibilité liée à l'énumération considérée. La chaîne de visibilité $a_4 \leq_{\mathsf{vis}} a_4$ est à la fois liée à l'énumération $\langle p_G, p_A\rangle$, et $\langle p_G, p_B\rangle$ étant donné que a_4 invite p_G. La chaîne de visibilité $m_1 \leq_{\mathsf{vis}} a_4$ est liée à l'énumération de pairs $\langle p_M, p_A\rangle$. La chaîne de visibilité $m_1 \leq_{\mathsf{vis}} b_1$ est liée à l'énumération de pairs $\langle p_M, p_B\rangle$. La chaîne de visibilité $a_4 \leq_{\mathsf{vis}} a_4 \leq_{\mathsf{vis}} a_4$ est à la fois liée à l'énumération $\langle p_G, p_M, p_A\rangle$ et à l'énumération $\langle p_G, p_M, p_B\rangle$, étant donné que a_4 invite p_G. La chaîne de visibilité $m_1 \leq_{\mathsf{vis}} a_4 \leq_{\mathsf{vis}} a_4$ est à la fois liée à l'énumération $\langle p_M, p_G, p_A\rangle$, et à l'énumération $\langle p_M, p_G, p_B\rangle$, étant donné que m_1 est une opération exécutée par p_M, a_4 invite p_G, et les pairs p_A et p_B observent m_1. Toutes ces chaînes débutent par m_1 ou a_4. a_1, a_2, a_3, et m_1 sont visibles ou égales à m_1 et a_4. Elles sont également observées par les pairs honnêtes invités. Ces quatres opérations sont donc stables.

Théorème 4.9 (Modèle de cohérence VFJC dynamique stabilisable)**.** *Le modèle de cohérence View-Fork-Join-Causal dynamique est stabilisable.*

Démonstration. Le modèle de cohérence View-Fork-Join-Causal dynamique repose sur le modèle de cohérence View-Fork-Join-Causal. D'après le théorème 4.9 toute opération d'une exécution abstraite qui respecte le modèle de cohérence View-Fork-Join-Causal est à terme stable. Donc toute opération d'une exécution abstraite qui respecte le modèle de cohérence *DynVFJC* est à terme stable. □

La stabilité View-Fork-Join-Causal dynamique identifie des opérations stables malgré la présence de pairs malintentionnés et l'évolution de la composition du groupe de pairs. Des opérations peuvent devenir stables alors que de nouveaux pairs sont invités au fur et à mesure de l'évolution de la collaboration. Nous pensons que le concept de stabilité peut trouver de nombreuses applications. Dans la section section 4.3 nous explorons l'une de ses applications : la troncature d'un journal infalsifiable et l'authentification de l'état d'une copie du contenu partagé à l'aide d'un journal tronqué infalsifiable.

4.2 Protocole à journaux complets

Nous présentons dans cette section un protocole qui assure la convergence des copies du contenu partagé malgré la présence de l'adversaire décrit dans le chapitre 2. Il repose sur le maintien d'un journal répliqué qui est conservé tout au long de la collaboration. Dans la section 4.3, nous nous baserons sur ce protocole pour construire un protocole qui permet la troncature des journaux et l'authentification d'états à partir de journaux tronqués.

Dans un premier temps nous présentons les structures de données principales qui sont nécessaires au fonctionnement du protocole. Nous présentons ensuite le protocole et des optimisations possibles de ce dernier. Nous montrons finalement que le protocole produit des journaux dans lesquels nous pouvons identifier des messages stables.

4.2.1 Structures de données

Un protocole repose sur des structures de données. Nous détaillons dans cette sous-section deux structures de données majeures sur lesquelles repose notre protocole : le journal et la copie du contenu partagé. Nous détaillons également les structures de données qui sont maintenues à jour avec le journal.

Copie du contenu partagé

Les pairs exécutent des opérations pour modifier et interroger le contenu partagé. Ils exécutent ces dernières sur leur copie S du contenu partagé. Les opérations d'interrogation et les opérations de modification dépendent du type du contenu.

Les protocoles que nous décrivons nécessitent la présence d'une *opération d'invitation* et d'une *opération d'acquiescement*. L'opération d'invitation permet à un nouveau pair de rejoindre la collaboration. L'opération d'acquiescement est directement exécutée par le protocole pour respecter certains invariants. Nous présenterons ces invariants lors de la description du protocole. L'opération d'acquiescement ne modifie pas le contenu partagé.

Tout type T de données répliquées peut être augmenté pour accepter ces deux opérations et ainsi donner un type G de données répliquées. Les opérations de G correspondent à celles de T et aux opérations d'invitation invite et d'acquiescement ack. Puisque nous n'avons pas ajouté d'opération d'interrogation, les valeurs de retour des exécutions des opérations d'interrogation de G sont identiques à celles de T.

$$\mathsf{Op}_G \stackrel{\text{def}}{=} \mathsf{Op}_T \cup \{\mathsf{ack}, \mathsf{invite}(q) \mid q \in \mathsf{Peers}\} \tag{4.1}$$

$$\mathsf{Val}_G \stackrel{\text{def}}{=} \mathsf{Val}_T \tag{4.2}$$

Chaque exécution d'une opération de modification de T, d'une opération d'invitation, ou d'une opération d'acquiescement génère un message. Un message contient les informations nécessaires à l'intégration de l'effet d'une opération quel que soit la copie sur laquelle il est intégré. Notre protocole garantit qu'un message est intégré exactement une fois dans un ordre qui respecte les causalités entre l'exécution des opérations. Un schéma de synchronisation par opérations est donc adéquate. Nous définissons donc le message

généré par l'exécution d'une opération d'invitation comme un n-uplet qui contient l'étiquette invite et le pair invité. L'exécution d'une opération d'acquiescement donne lieu à un message avec une étiquette ack.

$$\mathsf{Msg}_G \stackrel{\text{def}}{=} \mathsf{Msg}_T \cup \{\langle \mathsf{ack} \rangle, \langle \mathsf{invite}, q \rangle \mid q \in \mathsf{Peers}\} \tag{4.3}$$

Lorsqu'un pair p_i exécute une opération o, il dérive un message à partir de la fonction prepare. Il intègre ce message à sa copie du contenu répliqué, ainsi que tout autre message qu'il accepte, à l'aide de la fonction integrate. L'appel $\mathsf{prepare}_i(\sigma, o)$ retourne un message généré par l'exécution de o par le pair p_i sur l'état σ. L'appel $\mathsf{integrate}_i(\sigma, \mathsf{msg})$ retourne la mise-à-jour de l'état σ qui correspond à σ auquel a été intégré le message msg sur le pair p_i. Nous définissons l'état du contenu répliqué de type G comme un singleton qui contient un état de type T. Les opérations d'invitation et d'acquiescement ne modifient pas l'état. Les opérations de modification sont exécutées sur l'état de type T. Nous obtenons ainsi les définitions suivantes :

$$\mathsf{prepare}_i(\langle t \rangle, \mathsf{ack}) \stackrel{\text{def}}{=} \langle \mathsf{ack} \rangle \tag{4.4}$$

$$\mathsf{prepare}_i(\langle t \rangle, \mathsf{invite}(q)) \stackrel{\text{def}}{=} \langle \mathsf{invite}, q \rangle \tag{4.5}$$

$$\mathsf{prepare}_i(\langle t \rangle, o) \stackrel{\text{def}}{=} \mathsf{prepare}_i(t, o) \tag{4.6}$$

$$\mathsf{integrate}_i(\langle t \rangle, \langle \mathsf{ack} \rangle) \stackrel{\text{def}}{=} \langle t \rangle \tag{4.7}$$

$$\mathsf{integrate}_i(\langle t \rangle, \langle \mathsf{invite}, q \rangle) \stackrel{\text{def}}{=} \langle t \rangle \tag{4.8}$$

$$\mathsf{integrate}_i(\langle t \rangle, m) \stackrel{\text{def}}{=} \mathsf{integrate}_i(t, m) \tag{4.9}$$

L'exécution d'une opération d'interrogation par un pair p_i est simplement exécutée sur le contenu de type T.

$$\mathsf{eval}_i(\langle t \rangle, o) \stackrel{\text{def}}{=} \mathsf{eval}_i(t, o) \tag{4.10}$$

Tout type de données répliquées peut ainsi être augmenté pour être adapté à notre protocole. Pour illustrer notre protocole nous utilisons l'un des types de données répliquées les plus simples : un ensemble répliqué synchronisé par opérations dans lequel des valeurs peuvent seulement être ajoutées [13]. La figure 4.12 présente son implémentation.

$$\mathsf{OpGSet}\langle V \rangle \stackrel{\text{def}}{=} \mathcal{P}_{\mathsf{fin}}(V) \tag{4.11}$$

$$\mathsf{eval}_i(\sigma, \mathsf{rd}) \stackrel{\text{def}}{=} \sigma \tag{4.12}$$

$$\mathsf{prepare}_i(\sigma, \mathsf{add}(v)) \stackrel{\text{def}}{=} \langle \mathsf{add}, v \rangle \tag{4.13}$$

$$\mathsf{integrate}_i(\sigma, \langle \mathsf{add}, v \rangle) \stackrel{\text{def}}{=} \sigma \cup \{v\} \tag{4.14}$$

FIGURE 4.12 – Implémentation d'un ensemble répliqué synchronisé par opérations sans opération de suppression.

Journal de messages

L'exécution d'une opération de modification sur le contenu partagé dérive un message qui synthétise son effet. Les pairs échangent et intègrent les messages qu'ils produisent pour synchroniser leurs copies. Un pair enregistre l'ensemble des messages qu'il transmet et qu'il accepte des autres pairs dans son journal J. Un journal est un ensemble ordonné dont l'ordre entre les messages correspond à l'ordre dans lequel ils ont été ajoutés dans le journal. Cet ordre correspond donc également à l'ordre dans lequel les messages ont été intégrés à la copie du contenu partagé. Le journal est initialement vide.

Pour former un journal infalsifiable, chaque message est accompagné d'une signature cryptographique et référence éventuellement d'autres messages. Les messages qu'il référence sont ses *dépendances déclarées*. Tout message m est augmenté des méta-données suivantes :

peer_m **:** l'identifiant de l'auteur de m

num_m **:** le numéro de m

deps_m **:** l'ensemble des dépendances déclarées de m

sig_m **:** la signature cryptographique de m

Un pair honnête produit une séquence de messages avec des numéros contigus. Le premier message a un numéro égal à 1, le second message a un numéro égal à 2, et ainsi de suite. Le numéro d'un message m correspond donc au nombre de messages générés (m inclus) par son auteur peer_m. Un message m est identifié par un n-uplet dénoté $\mathsf{id}(m)$ composé de l'identifiant du pair qui est à son origine, son numéro, et son *hash* $h(m)$.

$$\mathsf{id}(m) \stackrel{\text{def}}{=} \langle \mathsf{peer}_m, \mathsf{num}_m, h(m) \rangle \tag{4.15}$$

Le *hash* permet de distinguer deux messages transmis par le même pair avec le même numéro, mais qui divergent par leur contenu ou leurs dépendances. Seuls les pairs malintentionnés sont à l'origine de tels messages. Le *hash* d'un message est calculé sur l'ensemble du message et de ses méta-données, à l'exception de sa signature. Nous supposons que la *fonction de hash* utilisée résiste aux *collisions*[13] et aux *préimages*[14].

Un message m dépend directement et indirectement d'un ensemble de messages. Un message déclare ses dépendances directes et éventuellement des dépendances indirectes[15]. deps_m correspond à l'ensemble des identifiants des dépendances déclarées de m.

Nous définissons la fonction $\mathsf{ctx}_J(m)$ pour extraire le sous-journal qui inclut uniquement l'ensemble des dépendances directes et transitives de m dans le journal J. Ce sous-journal n'inclut pas m. Cette fonction récursive termine, étant donné que le journal est un graphe dirigé fini sans cycle et que chaque appel récursif passe en paramètre un message qui correspond à une dépendance déclarée du message passé en paramètre de l'appel courant.

$$\mathsf{ctx}_J(m) \stackrel{\text{def}}{=} \{x \mid \exists y \in J.\, \mathsf{id}(y) \in \mathsf{deps}_m \wedge x \in \{y\} \cup \mathsf{ctx}_J(y)\} \tag{4.16}$$

Les collaborateur·ice·s disposent d'un couple de clefs asymétriques. Ils conservent leur clef privée et publient leur clef publique. Une clef publique permet d'identifier un·e

13. L'adversaire n'a pas les ressources nécessaires pour produire deux messages avec le même *hash*.
14. L'adversaire n'a pas les ressources nécessaires pour produire un message avec un *hash* donné.
15. Nous exploitons cette flexibilité pour optimiser certaines vérifications au sein du protocole présenté.

collaborateur·ice. Un·e collaborateur·ice peut être associé·e à plusieurs pairs. Chaque pair peut par exemple être un appareil distinct. L'identifiant d'un pair est un couple constitué d'une clef publique et d'un *hash*. Le *hash* permet de distinguer deux pairs qui disposent de la même clef publique. Pour s'assurer que le pair est invité une unique fois, le *hash* est généré à partir d'une partie du contenu du message d'invitation et de ses méta-données. Ce *hash* est calculé sur la clef publique du pair et sur l'ensemble des méta-données du message d'invitation à l'exception de sa signature. Nous supposons que la *fonction de hash* résiste aux *collisions* et aux *préimages*. Peers est l'ensemble des identifiants des pairs.

Nous représentons un journal à l'aide d'un graphe dirigé et linéarisé. Un sommet correspond à un message et un arc dirigé indique que sa source est une dépendance déclarée de sa destination. Par exemple, dans le journal de la figure 4.13, a_1 est une dépendance déclarée de a_2. Les messages sont horizontalement arrangés sur une ligne. Cet arrangement traduit leur ordre d'ajout dans le journal. Si un message est ajouté avant un autre message, alors le premier se trouve à gauche du second. Par exemple, le message a_1 est ajouté avant le message a_2. Le **??** indique le contenu et certaines méta-données des message du la figure 4.13. Les désignations des messages sont construites sur leurs identifiants. Ainsi un message généré par le pair p_i avec un numéro égal à k a pour désignation i_k. a_1 est donc un message généré par le pair p_A avec un numéro égal à 1. Les messages m_1 et m'_1 sont générés par le pair p_M avec un numéro égal à 1.

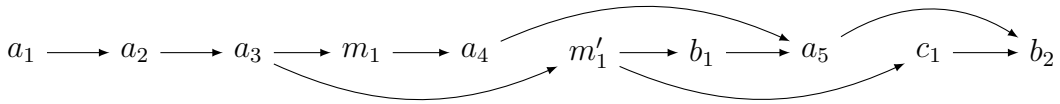


FIGURE 4.13 – Exemple du journal d'un pair.

message m	contenu de m	peer_m	num_m	deps_m
a_1	$\langle \mathsf{invite}, p_A \rangle$	p_A	1	$\emptyset$
a_2	$\langle \mathsf{invite}, p_M \rangle$	p_A	2	$\{a_1\}$
a_3	$\langle \mathsf{invite}, p_B \rangle$	p_A	3	$\{a_2\}$
m_1	$\langle \mathsf{add}, 1 \rangle$	p_M	1	$\{a_3\}$
a_4	$\langle \mathsf{ack} \rangle$	p_A	4	$\{m_1\}$
m'_1	$\langle \mathsf{invite}, p_C \rangle$	p_M	1	$\{b_1\}$
b_1	$\langle \mathsf{add}, 2 \rangle$	p_B	1	$\{m'_1\}$
a_5	$\langle \mathsf{ack} \rangle$	p_A	5	$\{a_4, b_1\}$
c_1	$\langle \mathsf{add}, 3 \rangle$	p_C	1	$\{m'_1\}$
b_2	$\langle \mathsf{add}, 4 \rangle$	p_B	2	$\{a_5, c_1\}$

TABLE 4.3 – Contenu et méta-données des messages du journal de la figure 4.13

Le journal de messages est la structure de données principale de notre protocole. Lorsqu'il est mis à jour d'autres structures de données que nous qualifions de *dépendantes* sont également mises à jour.

Structures de données dépendantes

Le journal J et la copie S du contenu partagé d'un pair p_i sont conservés en étroite synchronisation. Lorsqu'un message m est accepté dans le journal, m est intégré à la copie S. Cette intégration permet d'obtenir une version mise à jour de S notée S'. La copie S est initialisée avec un état prédéterminé par le protocole de réplication.

$$S' := \mathsf{integrate}_i(S, m) \tag{4.17}$$

Les messages d'un journal et leurs relations de dépendances forment un graphe dirigé sans cycle. max_J est l'ensemble des messages maximaux de J. Un message maximal de J n'est pas une dépendance déclarée d'un message de J. Lorsqu'un message m est ajouté dans le journal, l'ensemble est simplement mis à jour en ajoutant m et en retirant les messages qui sont des dépendances déclarées de m. Initialement cet ensemble est vide.

$$\mathsf{max}_{J'} := \{m\} \cup \{x \in \mathsf{max}_J \mid \mathsf{id}(x) \notin \mathsf{deps}_m\} \tag{4.18}$$

La restriction du graphe aux messages d'un pair honnête forme un ordre total strict, une « ligne ». Nous parlons ainsi de messages *linéaires*. Les messages d'un pair malintentionné peuvent former un arbre. Deux messages au sein d'*embranchements* distincts sont *non-linéaires*. Lorsqu'un journal J inclut des messages non-linéaires pour un pair p, nous disons que p est *reconnu malintentionné* dans J. $\mathsf{knownMalicious}_J$ est l'ensemble des pairs reconnus malintentionnés dans J. A chaque fois qu'un message non-linéaire est ajouté au journal, l'ensemble est mis à jour. Le prédicat $\mathsf{linear}_J(m)$ nous informe si le message est linéaire avec les messages déjà présents dans J. Le calcul du prédicat $\mathsf{linear}_J(m)$ est détaillé ci-après. $\mathsf{knownMalicious}_J$ est initialisé avec un ensemble vide.

$$\mathsf{knownMalicious}_{J'} := \mathsf{knownMalicious}_J \cup \begin{cases} \emptyset & \textbf{if } \mathsf{linear}_J(m) \\ \{\mathsf{peer}_m\} & \end{cases} \tag{4.19}$$

Nous notons vv_J un vecteur de versions [68, 69] qui associe à un pair le numéro du dernier message dont il est l'auteur qui a été ajouté dans J. Le numéro associé à un pair est initialisé à 0 lorsque son invitation est ajouté dans le journal. Lorsqu'un pair est reconnu malintentionné son association est supprimée du vecteur de versions. $\mathsf{vv}_J(p)$ retourne le numéro associé au pair p. Le vecteur de versions est initialement vide.

$$\begin{aligned} \mathsf{vv}_{J'} := & \{\langle p, \_\rangle \in \mathsf{vv}_J \mid p \neq \mathsf{peer}_m\} \\ & \cup \begin{cases} \{\langle q, 0\rangle\} & \textbf{if } m = \langle \mathsf{invite}, q\rangle \wedge \mathsf{peer}_m \neq q \\ \emptyset & \end{cases} \\ & \cup \begin{cases} \{\langle \mathsf{peer}_m, \mathsf{num}_m\rangle\} & \textbf{if } \mathsf{linear}_J(m) \\ \emptyset & \end{cases} \end{aligned} \tag{4.20}$$

Nous notons $\mathsf{invited}_J$, l'ensemble des pairs invités par les messages du journal J. Il est déduit des deux structures de données dépendantes précédentes.

$$\mathsf{invited}_J = \{p \mid \langle p, \_\rangle \in \mathsf{vv}_J\} \cup \mathsf{knownMalicious}_J \tag{4.21}$$

Un message bien-formé m est non-linéaire à un message d'un journal J si et seulement si il créée un embranchement ou si son auteur est déjà reconnu malintentionné dans J. Un message bien-formé dépend d'une suite de messages du même auteur avec des numéros contigus. L'ajout du message m dans J créée donc un nouvel embranchement si et seulement si il existe un autre message dans le journal du même auteur avec le même numéro. Si l'auteur du message est déjà reconnu malintentionné, alors le message m est forcément non-linéaire avec au moins un message du journal. Autrement, le message m est non-linéaire si son numéro n'est pas contigu avec celui du précédent message du même pair. Nous pouvons ainsi préciser le prédicat $\mathsf{linear}_J(m)$.

$$\begin{aligned}&\mathsf{linear}_J(m) \stackrel{\text{def}}{=} \\ &\quad \mathsf{peer}_m \notin \mathsf{knownMalicious}_J \wedge (\mathsf{peer}_m \in \mathsf{invited}_J \implies \mathsf{num}_m = \mathsf{vv}_J(\mathsf{peer}_m) + 1)\end{aligned} \tag{4.22}$$

Nous disposons des structures de données pour décrire notre protocole. Le tableau 4.4 présente l'évolution des structures de données dépendantes du journal de la figure 4.13.

ajout du message	S	max_J	$\mathsf{knownMalicious}_J$
	$\emptyset$	$\emptyset$	$\emptyset$
a_1	$\emptyset$	$\{a_1\}$	$\emptyset$
a_2	$\emptyset$	$\{a_2\}$	$\emptyset$
a_3	$\emptyset$	$\{a_3\}$	$\emptyset$
m_1	$\{1\}$	$\{m_1\}$	$\emptyset$
a_4	$\{1\}$	$\{a_4\}$	$\emptyset$
m'_1	$\{1\}$	$\{a_3, m'_1\}$	$\{p_M\}$
b_1	$\{1,2\}$	$\{a_3, b_1\}$	$\{p_M\}$
a_5	$\{1,2\}$	$\{a_5\}$	$\{p_M\}$
c_1	$\{1,2,3\}$	$\{a_5, c_1\}$	$\{p_M\}$
b_2	$\{1,2,3,4\}$	$\{b_2\}$	$\{p_M\}$

ajout du message	vv_J	$\mathsf{invited}_J$
	$\emptyset$	$\emptyset$
a_1	$\{\langle p_A, 1\rangle\}$	$\{p_A\}$
a_2	$\{\langle p_A, 2\rangle, \langle p_M, 0\rangle\}$	$\{p_A, p_M\}$
a_3	$\{\langle p_A, 3\rangle, \langle p_M, 0\rangle, \langle p_B, 0\rangle\}$	$\{p_A, p_M, p_B\}$
m_1	$\{\langle p_A, 3\rangle, \langle p_M, 1\rangle, \langle p_B, 0\rangle\}$	$\{p_A, p_M, p_B\}$
a_4	$\{\langle p_A, 4\rangle, \langle p_M, 1\rangle, \langle p_B, 0\rangle\}$	$\{p_A, p_M, p_B\}$
m'_1	$\{\langle p_A, 4\rangle, \langle p_B, 0\rangle, \langle p_C, 0\rangle\}$	$\{p_A, p_M, p_B, p_C\}$
b_1	$\{\langle p_A, 4\rangle, \langle p_B, 1\rangle, \langle p_C, 0\rangle\}$	$\{p_A, p_M, p_B, p_C\}$
a_5	$\{\langle p_A, 5\rangle, \langle p_B, 1\rangle, \langle p_C, 0\rangle\}$	$\{p_A, p_M, p_B, p_C\}$
c_1	$\{\langle p_A, 5\rangle, \langle p_B, 1\rangle, \langle p_C, 1\rangle\}$	$\{p_A, p_M, p_B, p_C\}$
b_2	$\{\langle p_A, 5\rangle, \langle p_B, 2\rangle, \langle p_C, 1\rangle\}$	$\{p_A, p_M, p_B, p_C\}$

TABLE 4.4 – Mises-à-jours successives des structures de données dépendantes du journal J après l'ajout successif des messages du **??**.

4.2.2 Description du protocole

Au cours de l'exécution d'un protocole, un pair est soumis à une succession d'événements qui font évoluer son état. Nous modélisons les pairs honnêtes comme des automates d'entrées-sorties soumis aux entrées suivantes :

— exécution d'une opération d'interrogation

— exécution d'une opération de modification

— livraison d'un message

— initialisation

Dans la suite de cette sous-section nous détaillons les étapes du protocole pour chacun de ces événements. La connaissance des structures de données décrites dans la sous-section précédente est nécessaire.

Exécution d'une opération d'interrogation

Lorsqu'un pair p_i souhaite interroger le contenu partagé, il exécute une opération d'interrogation sur sa copie S du contenu. Une opération d'interrogation o ne modifie pas la copie S du contenu partagé. De ce fait, elle n'induit pas la transmission d'un message aux autres pairs. Son évaluation sur S retourne une valeur rval.

$$\mathsf{rval} := \mathsf{eval}_i(S, o)$$

Exécution d'une opération de modification

Lorsqu'un pair p_i souhaite modifier le contenu partagé, il exécute une opération de modification o sur sa copie S du contenu. L'exécution d'une opération se déroule en plusieurs étapes. Le pair p_i dérive d'abord un message m à partir de l'opération o.

$$m := \mathsf{prepare}_i(S, o)$$

Le message m obtenu est ajouté dans le journal J. L'opérateur $++$ permet d'unir deux ensembles ordonnés. L'union conserve l'ordre respectif des éléments de chaque ensemble et produit un ordre entre les éléments du premier ensemble et les éléments du second ensemble : les éléments du premier ensemble sont plus petit que les éléments du second ensemble. L'union de deux ensembles ordonnés n'est pas commutatif.

$$J' := J ++ \langle m \rangle$$

Cet ajout met à jour l'ensemble des structures de données dépendantes du journal. C'est le cas notamment de la copie S qui intègre alors le message m.

Le calcul du *hash* de m nécessite ses méta-données. Dans les paragraphes suivants nous détaillons la construction des méta-données du message m.

Le pair se renseigne comme auteur du message.

$$\mathsf{peer}_m := p_i$$

Chaque message du pair a un numéro unique et contigu au numéro du message qu'il a précédemment généré. Le pair maintient un compteur ctr qu'il incrémente pour chaque nouveau message qu'il produit. Le compteur est initialisé à 1. Le premier message a donc un numéro égal à 1.

$$\mathsf{num}_m := \mathsf{ctr}$$
$$\mathsf{ctr}' := \mathsf{ctr} + 1$$

Le pair déclare les messages maximaux du journal comme dépendances de son message. Ce sont des messages qui ne sont pas des dépendances d'autres messages du journal.

$$\mathsf{deps}_m := \{\mathsf{id}(x) \mid x \in \mathsf{max}_J\}$$

Le pair signe son message à l'aide de la fonction sign. L'appel $\mathsf{sign}_i(\mathsf{data}_1 \parallel \mathsf{data}_2)$ concatène data_1 et data_2 puis signe le résultat avec la clef privée du pair p_i. Le contenu du message ainsi que ses méta-données sont signés.

$$\mathsf{sig}_m \stackrel{\text{def}}{=} sign_i(\mathsf{peer}_m \parallel \mathsf{num}_m \parallel \mathsf{deps}_m \parallel m)$$

Le message est finalement transmis à l'ensemble des pairs.

m dépend transitivement sur l'ensemble des messages du journal J. Ce message donne aux autres pairs une preuve que l'auteur de m connaît les pairs invités en m et dans ses dépendances. $\mathsf{ackInvited}$ est l'ensemble des pairs invités que p_i reconnaît. L'ensemble est initialement vide. Il est mis à jour après l'exécution d'une opération de modification.

$$\mathsf{ackInvited} := \mathsf{invited}_{J'}$$

Livraison d'un message

Les pairs s'échangent leurs messages sur un *réseau asynchrone corrompu* par l'adversaire. L'adversaire peut altérer, réordonner, omettre, retarder, et dupliquer des messages. Pour simplifier la description du protocole et nous concentrer sur ses points les plus importants, nous supposons qu'une couche réseau assure que (i) un message est livré exactement une fois, et que (ii) l'ensemble des dépendances d'un message sont préalablement livrés. Les messages des pairs honnêtes sont à terme tous livrés par cette couche réseau.

L'acceptation d'un message dans le journal J passe par deux étapes principales. Un pair vérifie d'abord que le message est bien-formé. Si le message est bien-formé, alors il est ensuite traité. Un message traité est soit accepté immédiatement dans le journal J ou il est mis en attente pour y être accepté ultérieurement sous certaines conditions.

Message bien-formé. L'adversaire du système peut produire des messages avec une structure cohérente, mais qui s'avèrent mal-formés. Les messages mal-formés ne sont pas acceptés par les pairs honnêtes. Un pair effectue plusieurs vérifications pour s'assurer qu'un message est bien-formé.

Il s'assure que le message m est authentique, c'est-à-dire que le message a bien été signé par le pair qui est déclaré comme son auteur. Pour ce faire, il dispose de la clef publique de chaque pair, du message, et des méta-données du message.

Le pair s'assure que le message fait partie de l'ensemble des messages admis par le protocole de réplication. Nous dénotons G le type de données répliquées de la copie S. Ce type admet notamment les opérations d'invitation et l'opérations d'acquiescement.

$$m \in \mathsf{Msg}_G \tag{4.23}$$

Si m est le premier message du journal, alors il s'agit d'une invitation qui invite son propre auteur. Si le message m est une invitation qui invite son propre auteur, alors il s'agit de la première opération du journal.

$$J = \emptyset \iff m = \langle \mathsf{invite}, \mathsf{peer}_m \rangle \tag{4.24}$$

Si le message m est une invitation, alors l'identifiant du pair est correctement généré. En d'autres termes, il s'agit d'un couple dont le premier élément est une clef publique et le second élément est un *hash* obtenu à partir de la clef publique et les méta-données du message à l'exception de sa signature.

Le pair s'assure que les dépendances déclarées du message m sont des messages bien-formés. Les messages bien-formés sont soit présent dans le journal J ou en attente. Le journal J étendu avec les messages en attente forme le *journal étendu* W. Le pair s'assure donc que chaque dépendance de m est présente dans le journal étendu W.

$$d \in \mathsf{deps}_m \implies \exists x \in W.\, d = \mathsf{id}(x) \tag{4.25}$$

Le pair s'assure ensuite que le message dépend transitivement d'un message du même auteur avec un numéro contigu. En d'autres termes, il ne dépend pas d'un message qui a un numéro égal ou plus grand que le sien.

$$\mathsf{num}_m = 1 + \mathsf{max}(\{\mathsf{num}_x \mid x \in \mathsf{ctx}_W(m) \wedge \mathsf{peer}_x = \mathsf{peer}_m\}) \tag{4.26}$$

Le pair s'assure que m dépend d'un message qui invite son auteur. Si m dépend d'un message du même auteur qui a déjà été accepté, alors m dépend d'un message qui invite son auteur. Ainsi nous devons vérifier si m dépend d'une invitation seulement lorsque son numéro est égal à 1.

$$\mathsf{num}_m = 1 \implies \exists x \in \mathsf{ctx}_W(m).\, x = \langle \mathsf{invite}, \mathsf{peer}_m \rangle \tag{4.27}$$

Finalement, le pair s'assure que l'auteur du message m et les auteurs des dépendances déclarées de m ne sont pas reconnus malintentionnés dans le sous-journal qui contient uniquement m et ses dépendances. Le sous-journal ne doit pas contenir deux messages avec le même auteur et le même numéro.

$$\forall \langle \mathsf{p}, \_, \_ \rangle \in \mathsf{deps}_m \cup \{\mathsf{id}(m)\}\ \nexists x, y \in \mathsf{ctx}_W(m).\, x \neq y \wedge p = \mathsf{peer}_x = \mathsf{peer}_y \wedge \mathsf{num}_x = \mathsf{num}_y \tag{4.28}$$

Remarque. Les pairs malintentionnés peuvent déclarer des dépendances indirectes arbitraires. Des messages théoriquement bien formés si on se restreint aux dépendances directes peuvent ainsi se retrouver mal-formés. Ceci n'a aucune incidence sur la convergence des copies, étant donné que l'ensemble des pairs honnêtes effectuent les mêmés vérifications et rejettent donc les même messages.

Traitement d'un message bien-formé. Si un message m est bien-formé, le pair doit déterminer s'il accepte immédiatement le message dans son journal J ou s'il le met en attente dans un ensemble ordonné $\mathsf{pending}$. Similairement au journal, l'ordre de l'ensemble $\mathsf{pending}$ correspond à l'ordre d'ajout des messages. L'ensemble $\mathsf{pending}$ est initialement vide. Le journal étendu W correspond au journal J auquel a été ajouté les messages de l'ensemble $\mathsf{pending}$ dans leur ordre d'apparition.

$$W \stackrel{\text{def}}{=} J +\!\!+ \mathsf{pending}$$

Le message m est accepté immédiatement dans le journal si et seulement si (i) le journal est vide, ou si (ii) le message est linéaire, les invitations de son auteur et des auteurs de ses dépendances sont présentes dans le journal J. Autrement, le message est mis en attente : il est ajouté à l'ensemble $\mathsf{pending}$. Il pourra être accepté ultérieurement, lorsqu'un message qui dépend de ce dernier est accepté dans le journal. Le prédicat $\mathsf{accepted}_J(m)$ indique si le message m peut être accepté immédiatement dans le journal J.

$$\begin{aligned} &\mathsf{accepted}_J(m) \iff \\ &\quad J = \emptyset \vee (\mathsf{linear}_J(m) \wedge \forall x \in \mathsf{ctx}_{\mathsf{pending}}(m).\, \mathsf{peer}_x, \mathsf{peer}_m \in \mathsf{invited}_J) \end{aligned} \tag{4.29}$$

Lorsqu'un message m est accepté dans le journal, l'ensemble de ses dépendances en attente sont également acceptées dans le journal. m est ajouté dans le journal après avoir ajouté ces messages en attente dans l'ordre dans lequel ils apparaissent dans $\mathsf{pending}$. Les messages en attente qui sont ajoutés dans le journal sont supprimés de l'ensemble $\mathsf{pending}$.

La mise-à-jour du journal est en réalité un peu plus complexe que l'équation présentée ci-dessus. En effet, le protocole exécute dans certains cas une opération d'acquiescement avant l'ajout d'un message et de ses dépendances en attente. Nous expliquons dans les deux paragraphes suivants dans quels cas cette exécution se produit.

Cas 1. Un pair honnête doit uniquement générer des messages bien-formés. Lorsqu'un pair génère un nouveau message m, il déclare comme dépendances de ce message les messages maximaux du journal ($\mathsf{deps}_m := \mathsf{max}_J$). Pour être bien-formé, ce message ne doit pas déclarer en tant que dépendance, un message dont l'auteur est reconnu malintentionné dans le journal. Il se peut que l'acceptation immédiate d'un message dans le journal conduise à l'ajout de messages en attente qui sont non-linéaires avec l'un des messages maximaux du journal. Un ou plusieurs auteurs de messages maximaux du journal peuvent donc être reconnu malintentionnés à l'issue de l'ajout dans le journal de messages en attente. Si l'ajout d'un message et de ses dépendances en attente conduisent à cette situation, alors le protocole exécute une opération d'acquiescement avant leurs ajouts. Pour simplifier, le protocole peut exécuter systématiquement une opération d'acquiescement avant l'ajout d'un message qui a au moins une dépendance en attente.

Cas 2. Un message qui dépend directement ou transitivement d'un message dont l'auteur n'est pas un invité connu du journal J n'est pas accepté immédiatement dans le journal. Il est mis en attente. Pour accepter les messages d'un nouvel invité, il est nécessaire qu'au moins un invité connu du journal accepte l'invitation de ce nouveau pair. Pour s'assurer qu'un de ses messages ne soit pas mis en attente, le pair doit donc s'assurer que les invitations des auteurs des dépendances de son message soient des invités connus

dans les autres journaux. Pour ce faire, il donne une preuve d'observation de l'invitation d'un pair avant d'accepter les messages de ce pair. Si l'auteur de m ou de l'une de ses dépendances en attente n'est pas reconnu par p_i ($p \notin \mathsf{ackInvited}$), alors le protocole exécute une opération d'acquiescement avant l'ajout de m et de ses dépendances en attente.

En considérant la simplification proposée pour le premier cas, une opération d'acquiescement est donc exécutée avant l'ajout d'un message m si et seulement si m a au moins une dépendance en attente ou son auteur n'est pas un invité connu. L'Algorithme 1 synthèse la livraison d'un message.

Algorithme 1 Livraison d'un message

```
1:  procedure deliver(m)
2:      if wellFormed(m) then                          ▷ le message m est-il bien-formé ?
3:          if accepted_J(m) then
4:              if ctx_pending(m) ≠ ∅ ∨ peer_m ∉ ackInvited then
5:                  execute(ack)                       ▷ exécuter une opération d'acquiescement
6:              end if
7:              J := J ++ ctx_pending(m) ++⟨m⟩
8:              pending := pending − ctx_pending(m)
9:          else
10:             pending := pending ++⟨m⟩
11:         end if
12:     end if
13: end procedure
```

Initialisation

Un pair peut initier une nouvelle collaboration ou en rejoindre une existante.

Initier une collaboration. Pour initier une nouvelle collaboration, un pair exécute une première opération qui l'invite. L'exécution de cette invitation génère son identifiant de pair.

Rejoindre une collaboration. Pour rejoindre une collaboration, un nouveau pair doit d'abord se faire inviter. Un pair de la collaboration peut inviter à tout moment un nouveau pair. Pour ce faire, il exécute une opération d'invitation qui génère l'identifiant p_i du nouveau pair. Il signe et transmet au nouveau pair un message qui contient l'identifiant p_i et son journal. Un pair malintentionné peut transmettre un identifiant quelconque et un journal arbitraire.

Le nouveau pair p_i vérifie d'abord l'authenticité du message. Il construit son journal J et l'état de sa copie S à partir du journal reçu. Lors de la construction de son journal, il vérifie que le journal reçu a été correctement construit. L'Algorithme 2 synthèse l'initialisation du journal et de la copie d'un pair p_i à partir d'un journal reçu G. L'instruction **check** permet de vérifier qu'une condition est remplie. Si la condition n'est pas remplie l'initialisation échoue. Nous détaillons cet algorithme étape par étape dans les paragraphes suivants.

Algorithme 2 Initialisation à partir d'un journal reçu G

```
1:  procedure init_i(G)
2:      ackInvited := ∅
3:      let ⟨y_1, ..., y_n⟩ := G
4:      let k := 1
5:      while k ≤ n do
6:          check wellFormed(y_k)
7:          if accepted_J(y_k) then
8:              let msgs := ctx_pending(y_k) ++ ⟨y_k⟩
9:              if p_i ∈ ackInvited ∧ ∃x ∈ msgs. peer_x ∉ ackInvited then
10:                 execute(ack)                ▷ exécuter une opération d'acquiescement
11:             end if
12:             J := J ++ msgs
13:             pending := pending − msgs
14:             if ⟨invite, p_i⟩ ∈ msgs then    ▷ p_i est-il invité par l'un des messages ?
15:                 check max_J = {⟨invite, p_i⟩}
16:                 ackInvited := invited_J
17:             end if
18:             check ∀x ∈ max_J. peer_x ∉ knownMalicious_J
19:         else
20:             pending := pending ++ ⟨y_k⟩
21:         end if
22:         k := k + 1
23:     end while
24:     check pending = ∅ ∧ p_i ∈ invited_J
25: end procedure
```

Le pair vérifie que chaque message est bien-formé et traite les messages dans l'ordre dans lequel ils ont été ajoutés dans le journal reçu. Le traitement d'un message suit les mêmes étapes que celles présentées pour la livraison d'un message à une exception près : l'exécution d'une opération d'acquiescement avant l'acceptation immédiate d'un message est soumise à des conditions plus strictes.

Lors de la livraison d'un message, un pair exécute une opération d'acquiescement avant l'acceptation immédiate d'un messages si et seulement si (i) le message a au moins une dépendance en attente, ou si (ii) l'auteur du message ou d'une dépendance en attente n'est pas un invité que le pair reconnaît. La première condition garantit l'absence de messages non-linéaires parmi les messages maximaux du journal. La seconde condition garantit que l'acceptation immédiate par les autres pairs honnêtes des message générés ultérieurement par le pair n'est pas conditionnée par l'acceptation de l'invitation d'un pair qu'il n'a pas préalablement reconnu.

Si le journal reçu est correctement construit, ses états successifs ne présentent pas de messages non-linéaires parmi leurs messages maximaux. Le nouveau pair n'a donc pas besoin d'exécuter d'opérations d'acquiescement lorsque la première condition est remplie. Il doit simplement vérifier que l'ajout de messages dans son journal ne conduit pas à la

présence de messages non-linéaires parmi les messages maximaux de son journal (ligne 18).

Les messages d'un pair dépendent tous du message qui l'invite. Une invitation contraint donc un pair à reconnaître comme invité lui-même ainsi que tous les pairs invités dans les dépendances de l'invitation. Lorsqu'il ajoute le message qui l'invite dans son journal, il initialise l'ensemble des pairs qu'il reconnaît comme invité. Puisque le journal reçu est celui du pair qui invite le nouveau pair, le message qui invite le nouveau pair doit dépendre d'un préfixe du journal reçu et doit être accepté immédiatement (ligne 15). Les invités reconnus correspondent donc aux invités connus du journal après l'ajout de l'invitation (ligne 16).

Une fois que le message qui l'invite est ajouté dans son journal, le nouveau pair fait partie des invités connus du journal. Il peut alors exécuter des opérations d'acquiescement sans générer de messages mal-formés. Pour les mêmes raisons que celles évoqués précédemment, il exécute une opération d'acquiescement avant l'acceptation immédiate d'un message lorsqu'il ne reconnaît pas encore l'auteur de ce message ou de l'une de ses dépendances en attente (ligne 9 et ligne 10).

L'initialisation réussit si le pair fait partie des invités connus du journal et l'ensemble des messages du journal reçu figurent dans le journal du nouveau pair (ligne 24). Il n'y a pas de messages mal-formés et de message qui demeurent en attente.

4.2.3 Exemple d'une exécution du protocole

Pour illustrer le protocole à journaux complets, nous prenons une exécution dans laquelle quatre pairs modifient un ensemble répliqué. Les pairs p_A, p_B, et p_C sont honnêtes. Le pair p_M est un pair malintentionné. Le protocole est exécuté par les pairs honnêtes. La figure 4.14 et la figure 4.15 présentent respectivement l'évolution de l'état du journal du pair p_A et l'évolution de l'état du journal du pair p_B. Elles présentent également l'évolution de l'état de certaines structures de données. Ces structures de données devraient suffire à appréhender l'évolution de l'état du journal. Nous avons précédemment présenté dans la figure 4.13 l'état final du journal du pair p_A. L'évolution de l'état des autres structures de données du pair p_A peut donc être suivi dans le tableau 4.4. Le tableau 4.5 rappelle le contenu des messages et leurs méta-données.

Nous détaillons l'évolution de l'état du journal du pair p_A et l'évolution de l'état du journal du pair p_B dans les deux paragraphes qui suivent. Nous supposons que tous les messages livrés sont correctement signés.

Le pair p_A initie la collaboration (1). Il exécute ainsi une première opération qui l'invite lui-même. Cette exécution génère un message $a_1 \stackrel{\text{def}}{=} \langle \mathsf{invite}, p_A \rangle$. Le pair invite ensuite le pair p_M (2) et le pair p_B (3) et génère ainsi les messages $a_2 \stackrel{\text{def}}{=} \langle \mathsf{invite}, p_M \rangle$ et $a_3 \stackrel{\text{def}}{=} \langle \mathsf{invite}, p_B \rangle$. Il livre le message bien-formé m_1 (4). Il est accepté immédiatement dans le journal J. p_A livre ensuite le message bien-formé m'_1 (5). m'_1 n'est pas linéaire avec au moins un message du journal $\neg\, \mathsf{linear}_J(m'_1)$. En l'occurrence, il est non-linéaire avec le message m_1. m'_1 est donc mis en attente. p_A livre le message bien-formé c_1 (6). Il s'agit d'un message linéaire dans J. Cependant, son auteur n'est pas un invité reconnu dans J ($p_C \notin \mathsf{invited}_J$). c_1 est donc mis en attente. p_A livre le message bien-formé b_2 (7). Il est

accepté immédiatement dans le journal. Étant donné qu'il dépend du message en attente m'_1, ce dernier est également accepté dans le journal. Avant ces ajouts, le protocole exécute une opération d'acquiescement qui donne lieu à la génération du message $a_4 \stackrel{\text{def}}{=} \langle ack \rangle$. Cette exécution permet d'éviter d'avoir le message non-linéaire m_1 parmi les messages maximaux du journal. L'ajout de m'_1 dans le journal implique la reconnaissance de p_M comme un pair malintentionné. Toute livraison ultérieure d'un message bien-formé de p_M sera systématiquement mis en attente. p_C est désormais un invité connu dans J. Finalement, p_A livre le message bien-formé b_3 (8). b_3 est accepté immédiatement dans le journal. Puisqu'il dépend du message en attente c_1, ce dernier est également accepté dans le journal. L'auteur de c_1 n'est pas un invité reconnu par p_A. Avant d'ajouter c_1 et b_3, le protocole exécute une opération d'acquiescement qui donne lieu à la génération du message $a_5 \stackrel{\text{def}}{=} \langle \mathsf{ack} \rangle$.

Le pair p_B récupère un journal dans lequel il est invité pour initialiser son journal et les structures de données dépendantes de son journal (1). Il livre le message bien-formé $m'_1 \stackrel{\text{def}}{=} \langle \mathsf{invite}, p_C \rangle$ (2). Le message est accepté immédiatement dans le journal. L'ajout du message dans le journal met à jour les invités connus du journal J. p_C est un invité connu. En revanche il n'est pas encore reconnu par p_B. p_B livre ensuite $m_1 \stackrel{\text{def}}{=} \langle \mathsf{add}, 1 \rangle$ (3). m_1 est non-linéaire avec un message du journal. Il est donc mis en attente. p_B exécute une opération qui donne lieu à la génération du message $b_1 \stackrel{\text{def}}{=}= \langle \mathsf{add}, 3 \rangle$. Il reconnaît que p_C est invité. Il livre ensuite les messages bien-formés a_4 (5) et a_5 (6). Ces deux messages sont acceptés immédiatement dans le journal. a_4 dépend du message en attente m_1. m_1 est donc ajouté dans le journal avant l'ajout de a_4. L'ajout de m_1 et a_4 n'aboutit pas à la présence d'un message non-linéaire parmi les messages maximaux du journal. Le protocole n'exécute donc pas uen opération d'acquiescement avant ces ajouts. p_B livre le message bien-formé et linéaire c_1. L'auteur de c_1 est un invité connu du journal. c_1 est donc accepté immédiatement dans le journal. Puisque l'auteur de c_1 est reconnu par p_B ($p_C \in \mathsf{ackInvited}$), le protocole n'exécute pas d'opération d'acquiescement avant l'ajout du message. Finalement, p_B exécute une opération qui génère le message $b_2 \stackrel{\text{def}}{=} \langle \mathsf{add}, 4 \rangle$.

message m	contenu de m	peer_m	num_m	deps_m
a_1	$\langle \mathsf{invite}, p_A \rangle$	p_A	1	$\emptyset$
a_2	$\langle \mathsf{invite}, p_M \rangle$	p_A	2	$\{a_1\}$
a_3	$\langle \mathsf{invite}, p_B \rangle$	p_A	3	$\{a_2\}$
m_1	$\langle \mathsf{add}, 1 \rangle$	p_M	1	$\{a_3\}$
a_4	$\langle \mathsf{ack} \rangle$	p_A	4	$\{m_1\}$
m'_1	$\langle \mathsf{invite}, p_C \rangle$	p_M	1	$\{b_1\}$
b_1	$\langle \mathsf{add}, 2 \rangle$	p_B	1	$\{m'_1\}$
a_5	$\langle \mathsf{ack} \rangle$	p_A	5	$\{a_4, b_1\}$
c_1	$\langle \mathsf{add}, 3 \rangle$	p_C	1	$\{m'_1\}$
b_2	$\langle \mathsf{add}, 4 \rangle$	p_B	2	$\{a_5, c_1\}$

TABLE 4.5 – Contenu et méta-données des messages du journal de la figure 4.13

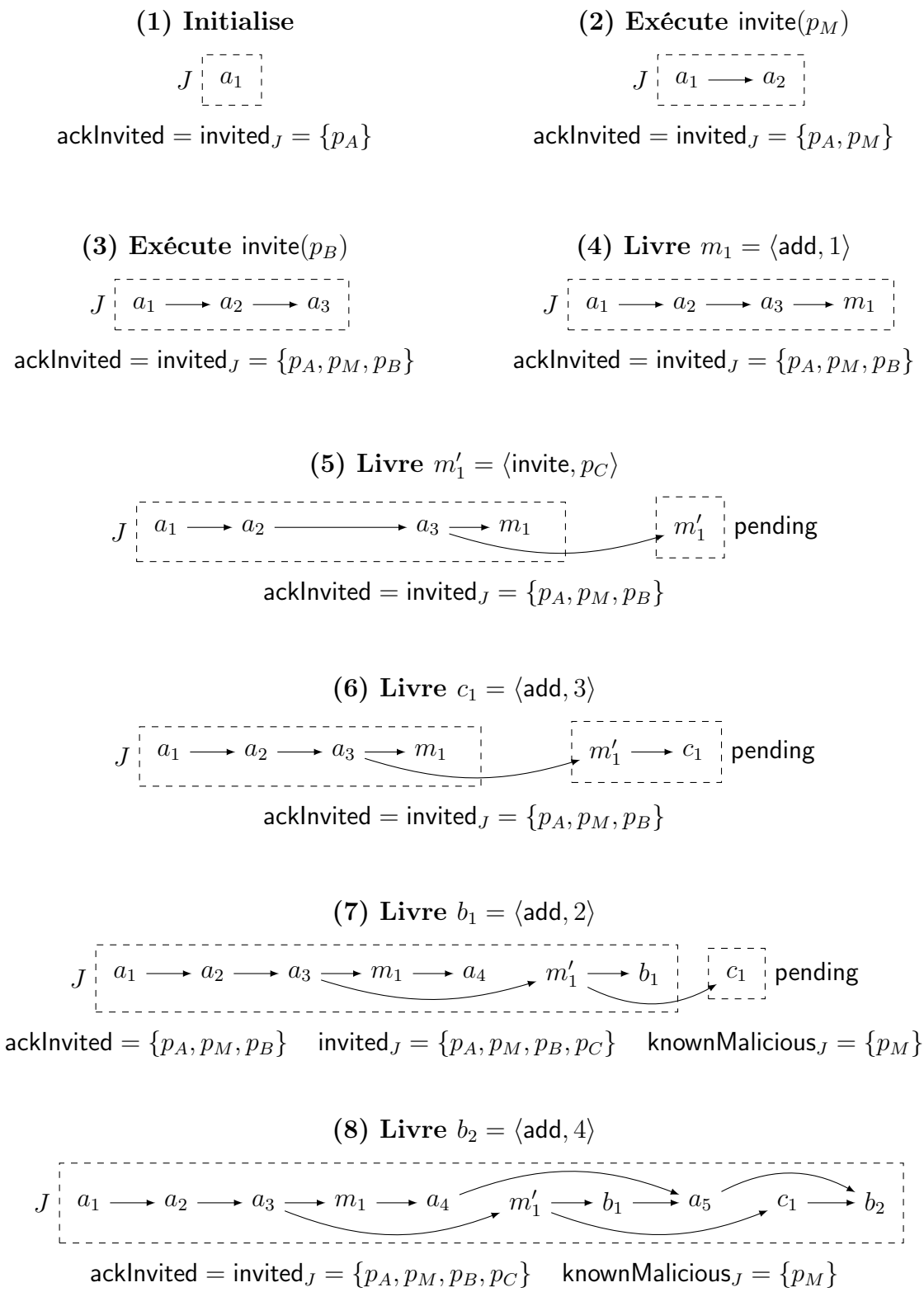


FIGURE 4.14 – Exemple d'exécution du protocole à journaux complets par un pair p_A. Voir le **??** pour plus de détails sur les messages.

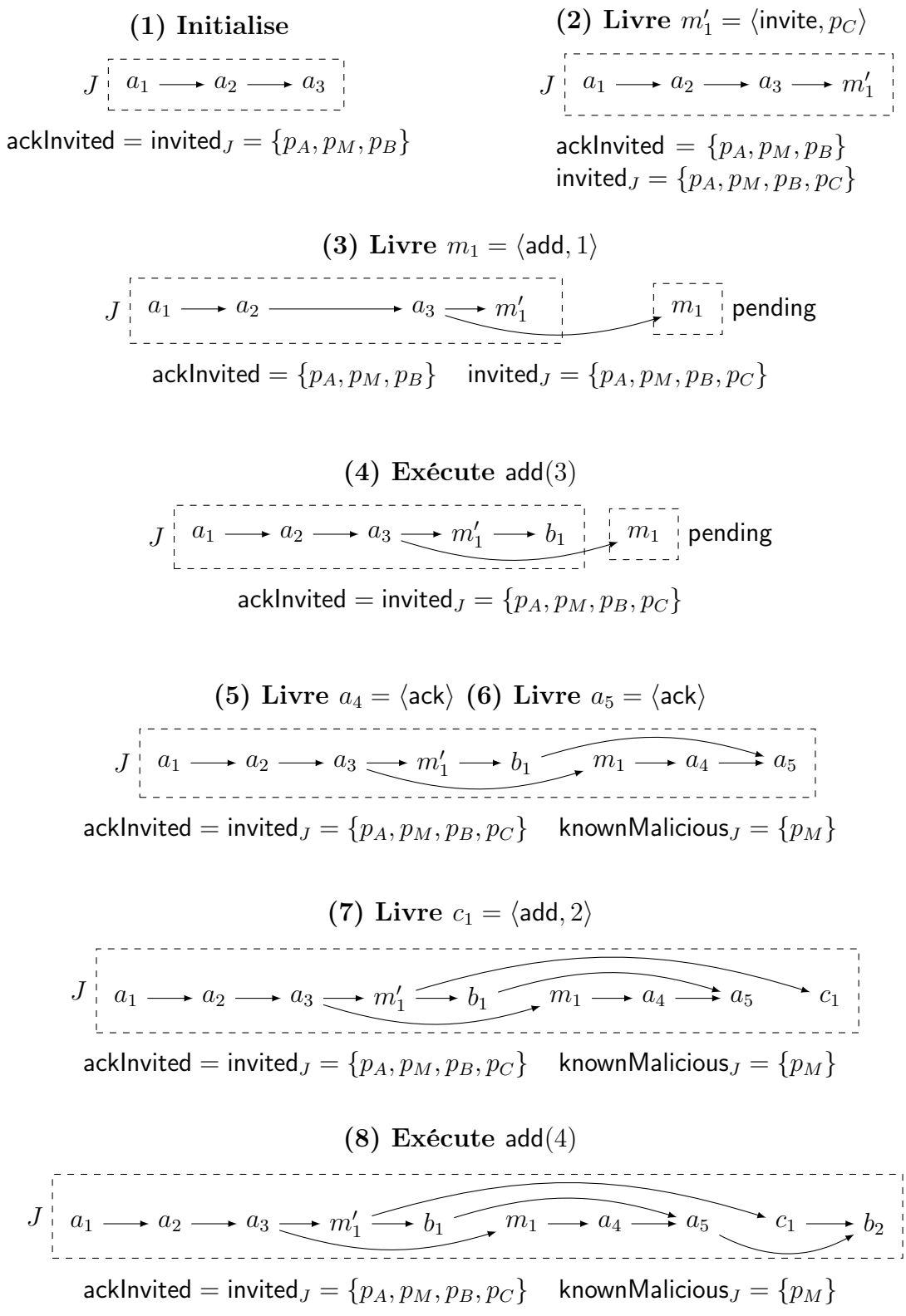


FIGURE 4.15 – Exemple d'exécution du protocole à journaux complets par un pair p_B. Voir le **??** pour plus de détails sur les messages.

4.2.4 Optimisations

Nous présentons dans cette sous-section des optimisations du protocole à journaux complets.

En pratique l'opération d'invitation et le message d'invitation pourraient contenir uniquement la clef publique du collaborateur ou de la collaboratrice invité·e. L'identifiant du pair correspondrait alors au *hash* du message. Notre approche permet d'avoir des opérations d'invitations avec une signature identique à celle présentée pour le modèle de cohérence *Invitation*. Ce modèle de cohérence est présenté dans la section 4.1.

Pour être bien-formé, un message m doit être linéaire et doit dépendre d'un message d'invitation qui invite l'auteur de m. Dans le protocole que nous avons présenté, ces deux vérifications requièrent le parcours du journal (équation 4.26 et équation 4.27). Nous pouvons réduire ces vérifications à une simple vérification sur les dépendances déclarées du message.E Pour ce faire, lorsqu'un pair génère un nouveau message, il inclut toujours son précédent message comme dépendance déclarée et à défaut le message qui l'invite. Pour être bien-formé un message doit inclure cette dépendance déclarée. Ainsi, si le message a un numéro égal à 1, il doit inclure une dépendance déclarée qui invite son auteur. Autrement, le message doit inclure une dépendance déclarée du même auteur avec un numéro égal à son numéro auquel 1 est soustrait.

$$\exists x \in W.\, \mathsf{id}(x) \in \mathsf{deps}_m \wedge \begin{cases} \mathsf{num}_m = \mathsf{num}_x + 1 & \textbf{if } \mathsf{num}_m > 1 \\ x = \mathsf{invite}(\mathsf{peer}_m) \end{cases} \tag{4.30}$$

Pour être bien-formé, un message m ne doit pas reconnaître comme malintentionné son auteur ou l'auteur d'une de ses dépendances déclarées. Cette vérification requiert le parcours du journal (équation 4.28). Nous pouvons utiliser un système de marquage qui permet de connaître les embranchements sur lesquels dépend un message. Tous les messages d'un embranchement sont associés à l'identifiant du message à l'origine de l'embranchement. Lorsqu'un message non-linéaire d'un pair donné est ajouté pour la première fois dans le journal, le journal est parcouru pour marquer les messages du premier embranchement. Lorsqu'un message est ajouté au journal il est associé aux marqueurs de ses dépendances déclarées. Ainsi, un message est bien-formé s'il n'hérite pas d'un marqueur dont son auteur est à l'origine et s'il n'hérite pas de deux ou plusieurs marqueurs du même pair. Étant donné que le nombre d'embranchements acceptables est borné [47], le nombre de parcours pour le marquage et le nombre de marqueurs associés à un message sont également bornés.

Pour rejoindre une collaboration, un nouveau pair récupère le journal du pair qui l'invite. En réalité tout pair peut transmettre son journal au nouveau pair. Le nouveau pair doit s'assurer que son invitation dépendent d'un préfixe du journal. Il peut modifier l'ordre topologique des messages au sein du journal reçu pour s'en assurer. Cette vérification est nécessaire pour éviter une attaque qui viserait à faire accepter au nouveau pair le message d'un auteur qu'il n'a pas préalablement reconnu. Ce qui pourrait conduire à la mise en attente par les autres pairs honnêtes des messages qu'il génère.

4.2.5 Cohérence du journal

Un pair accepte dans son journal des messages bien-formés et sous des conditions bien déterminées. De ce fait, les messages du journal et leurs relations respectent des contraintes. Dans cette section nous introduisons une correspondance entre journaux et exécutions abstraites. Une exécution abstraite est l'abstraction que nous avons utilisée pour définir les modèles de cohérence et le concept de stabilité. Nous montrons alors que l'exécution abstraite associée à un journal respecte notre modèle de cohérence View-Fork-Join-Causal Dynamique.

Nous avons précédemment défini un journal comme un ensemble ordonné. L'ordre entre les messages correspond à l'ordre d'ajout des messages dans le journal. Pour faciliter la mise en correspondance du journal avec une exécution abstraite, nous définissons dans cette sous-section un journal comme un n-uplet $\langle E, \mathsf{owner}, \mathsf{ao} \rangle$ tel que :

— E est l'ensemble dénombrable des messages du journal.

— $\mathsf{owner} \in \mathsf{Honest}$ est le pair honnête qui maintient le journal.

— $\mathsf{ao} \subseteq E \times E$ est l'ordre dans lequel les messages ont été ajoutés dans le journal. Il s'agit d'un ordre total strict localement fini.

Nous définissons la relation transitive suivante sur les messages du journal :

Dépendance. Une relation qui rend compte des dépendances directes et indirectes entre les messages d'un journal. Un message x est une dépendance d'un message z, et nous écrivons $x \xrightarrow{\mathsf{dep}} z$, si et seulement si x est une dépendance déclarée de z ou s'il existe une dépendance y de z telle que x est une dépendance de y.

$$x \xrightarrow{\mathsf{dep}} z \overset{\text{def}}{\iff} \mathsf{id}(x) \in \mathsf{deps}_z \vee \exists y.\, x \xrightarrow{\mathsf{dep}} y \xrightarrow{\mathsf{dep}} z$$

A chaque message d'un journal correspond une opération exécutée dans son exécution abstraite associée. Chaque message devrait donc permettre de caractériser une opération exécutée. Puisque nous souhaitons montrer que l'exécution abstraite associée au journal respecte le modèle de cohérence *DynVFJC*, seuls les messages d'invitations doivent être obligatoirement associés à une opération d'invitation. Les autres messages peuvent être associés à des opérations quelconques tels que l'opération d'acquiescement.

Une exécution abstraite rend compte de l'observation de chaque pair honnête. En déclarant comme dépendance l'un de ses messages, un pair honnête atteste avoir observé (et intégré) la dépendance avant l'exécution de l'opération à l'origine du message. Les relations de dépendance entre les messages exposent donc les observations des pairs honnêtes. Nous pouvons donc créer une correspondance entre les relations de dépendance au sein d'un journal et les relations de visibilité au sein de l'exécution abstraite associée au journal.

Les dépendances déclarées par les pairs malintentionnées ne traduisent pas nécessairement leurs observations réelles. Il en va de même au sein d'une exécution abstraite : les relations de *visibilité* ne traduisent pas les observations des pairs malintentionnées.

Cette correspondance n'est pas suffisante. Lorsqu'un pair honnête ajoute un message dans son journal, il l'observe. Cette observation devrait être exposée au sein de l'exécution abstraite associée au journal. Pour ce faire, une exécution abstraite associée inclut une

opération supplémentaire : *l'opération d'observation implicite*. L'opération d'observation implicite a pour auteur le possesseur du journal. Toutes les autres opérations sont visibles à l'opération d'observation implicite.

La Definition 4.12 définit l'exécution abstraite associée à un journal. La figure 4.16 donne un exemple d'exécution abstraite associée à un journal. $\top_{\mathsf{vis}}$ est l'opération d'observation implicite.

Definition 4.12 (Exécution abstraite associée à un journal)**.** Pour tout journal $J \stackrel{\text{def}}{=} \langle \_, \mathsf{owner}, \mathsf{ao} \rangle$, il existe une unique histoire associée $H \stackrel{\text{def}}{=} \langle \_, \mathsf{peer}, \mathsf{call}, \mathsf{rval}, \mathsf{rb} \rangle$ et une unique exécution abstraite associée $A \stackrel{\text{def}}{=} \langle H, \mathsf{vis} \rangle$. L'histoire H est munie d'une opération exécutée particulière $\top_{\mathsf{vis}}$ que nous nommons opération d'observation implicite. Il existe entre les messages du journal J et les opérations exécutées de l'exécution abstraite A privée de l'opération d'observation implicite $\top_{\mathsf{vis}}$ une correspondance un-à-un (une bijection). Pour un message x_J nous notons x_A l'opération exécutée correspondante. L'histoire H et l'exécution abstraite A sont construites à partir de cette correspondance telles que :

— L'auteur d'une opération x_A est le possesseur du journal si x_A est l'opération d'observation implicite sinon il s'agit de l'auteur de son message associé x_j.

$$\mathsf{peer}(x_A) \stackrel{\text{def}}{=} \begin{cases} \mathsf{peer}_{x_J} & \textbf{if } x_A \neq \top_{\mathsf{vis}} \\ \mathsf{owner} \end{cases}$$

— L'appel de l'opération x_A est une opération d'acquiescement si x_A est l'opération d'observation implicite. Si x_A n'est pas l'opération d'observation implicite., l'appel de x_A est une opération d'invitation si le message associé est un message d'invitation ou une opération d'acquiescement.

$$\mathsf{call}(x_A) \stackrel{\text{def}}{=} \begin{cases} \mathsf{invite}(q) & \textbf{if } x_A \neq \top_{\mathsf{vis}} \wedge x_J = \langle \mathsf{invite}, q \rangle \\ \mathsf{ack} \end{cases}$$

— La relation *retourne-avant* rb est construite à partir de l'ordre d'ajout des messages dans le journal. Toute opération du journal retourne-avant l'opération d'observation implicite $\top_{\mathsf{vis}}$.

$$x_A \xrightarrow{\mathsf{rb}} z_A \iff^{\text{def}} z_A = \top_{\mathsf{vis}} \vee x_J \xrightarrow{\mathsf{ao}} z_J$$

— La relation de visibilité vis est construite à partir des relations de dépendance entre les messages. Toute opération de A est visible ou égale à l'opération d'observation implicite $\top_{\mathsf{vis}}$.

$$x_A \xrightarrow{\mathsf{vis}} z_A \iff^{\text{def}} z_A = \top_{\mathsf{vis}} \vee x_J \xrightarrow{\mathsf{dep}} z_J$$

Nous démontrons dans la suite de cette sous-section que l'exécution abstraite associée à un journal respecte le modèle de cohérence *DynVFJC* à l'aide du théorème 4.10 et du théorème 4.11.

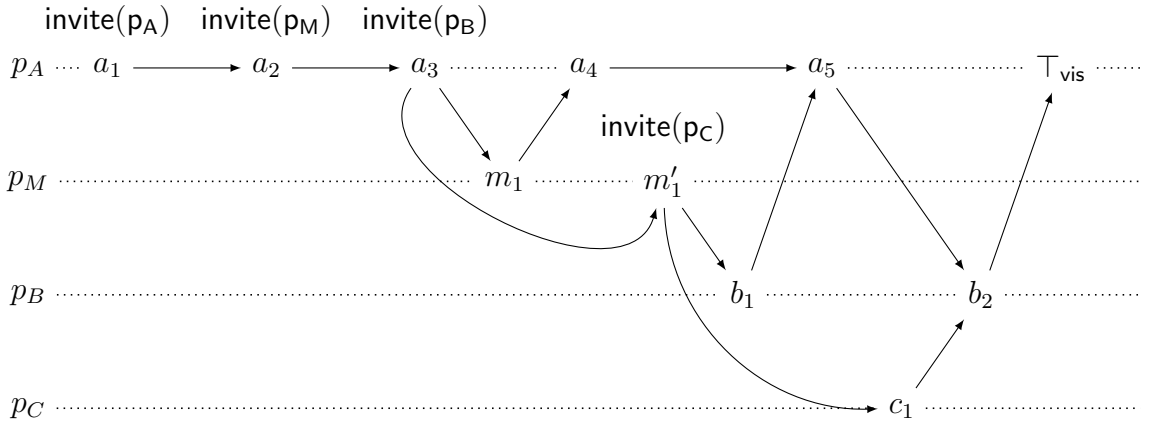


FIGURE 4.16 – Exécution abstraite associée au journal de la figure 4.13.

Théorème 4.10 (Exécution abstraite associée bien-formée)**.** *L'exécution abstraite associée à un journal est bien-formée.*

Pour nous aider dans la démonstration de ce théorème, nous démontrons le lemme suivant :

Lemme 4.10.1. *La relation* retourne-avant *de l'exécution abstraite associée à un journal est un ordre total strict.*

Démonstration. Si un message x_J est ajouté avant un message y_J dans le journal J alors l'opération x_A associée à x_J retourne-avant l'opération y_A associée à y_J dans l'exécution abstraite A associée à J. L'ordre d'ajout des messages dans le journal est un ordre total strict. L'opération d'observation implicite est la seule opération qui n'est pas associée à un message. Par définition toute les autres opérations retourne-avant l'opération d'observation implicite. La relation retourne-avant est donc un ordre total strict. □

Démonstration du théorème 4.10. Suivant le Lemme 4.10.1, nous déduisons que toute restriction de la relation *retourne-avant* aux opérations d'un pair donné (malintentionné ou honnête) est un ordre total strict. Par conséquent, la propriété BF1 est respectée.

Toute opération de A est visible ou égale à l'opération d'observation implicite $\top_{\mathsf{vis}}$. L'auteur de cette dernière est le possesseur du journal. Il est par définition honnête. Par conséquent, toutes les opérations, y compris les opérations des pairs malintentionnées, sont observées par au moins un pair honnête. La propriété BF2 est satisfaite. □

Théorème 4.11 (Exécution abstraite associée DynVFJC)**.** *Un journal J construit à partir du protocole à journaux complets est associé à une exécution abstraite A qui respecte le modèle de cohérence* DynVFJC*. A privée de l'opération d'observation implicite respecte également* DynVFJC*.*

Pour démontrer de ce théorème, nous démontrons plusieurs lemmes :

Lemme 4.11.1. *Lorsqu'un message m est ajouté dans le journal J du pair p_i, l'ensemble des dépendances déclarées sont déjà présentes dans le journal.*

Démonstration. Un message est ajouté dans le journal J lors de l'exécution d'une opération de modification, lors de la livraison d'un message, ou lorsque le pair rejoint la collaboration. Lors de l'exécution d'une opération de modification, un message est généré et ajouté dans le journal. Le message déclare en dépendances les messages de l'ensemble max_J. max_J inclut uniquement des messages déjà présents dans le journal. Lors de la livraison d'un message et lors du processus d'initialisation qui permet à un pair de rejoindre la collaboration, le pair vérifié que le message est bien-formé. Une des conditions pour être bien-formé est d'avoir l'ensemble de ses dépendances incluses dans le journal étendu W (équation 4.25). Lorsque m est ajouté dans le journal J, l'ensemble de ses dépendances présentes dans W et absente de J sont préalablement ajoutés dans J. □

Lemme 4.11.2. *La relation de dépendance entre les messages du journal est un ordre partiel strict et l'ordre d'ajout des messages dans le journal est une extension linéaire de cet ordre partiel.*

Démonstration. Par construction la relation de dépendance est transitive et stricte. Par construction l'ordre d'ajout des messages est total et strict. D'après le Lemme 4.11.1, les dépendances d'un message sont présentes dans le journal avant l'ajout de ce dernier. Par conséquent, il ne peut y avoir de cycle de dépendances. La relation de dépendance est donc un ordre partiel strict et l'ordre d'ajout en est une extension linéaire. □

Lemme 4.11.3. *La relation de dépendance restreinte aux messages du possesseur du journal forme un ordre total strict.*

Démonstration. Lorsque le pair exécute une opération, il dérive un message qui déclare comme dépendances les messages de l'ensemble max_J. Le message est alors ajouté dans cet ensemble. Il est supprimé de cet ensemble lorsqu'un message qui le déclare comme dépendance y est ajouté. S'il ne s'agit pas de son premier message, un pair déclare donc toujours une dépendance qui est son précédent message ou un message qui dépend transitivement de son précédent message. La relation de dépendance restreinte aux messages du possesseur du journal forme donc un ordre total strict. □

Lemme 4.11.4. *L'ensemble* max_J *des messages maximaux du journal* J *inclut uniquement des messages dont l'auteur n'est pas reconnu malintentionné dans* J*.*

Démonstration. Un message est ajouté dans le journal par l'exécution d'une opération de modification, la livraison d'un message bien-formé, ou lorsqu'un pair rejoint la collaboration.

L'exécution d'une opération de modification entraîne la génération d'un message m. Ce dernier déclare en dépendances l'ensemble des messages maximaux du journal. Lors de son ajout dans le journal, l'ensemble des messages maximaux de J est donc égal au singleton qui contient ce nouveau message $\mathsf{max}_J = \{m\}$. D'après le Lemme 4.11.3, les messages du possesseur du journal sont linéaires. A l'issue de l'exécution d'une opération de modification, l'ensemble max_J contient donc un seul message qui est non-linéaire.

La livraison d'un message bien-formé aboutit à l'acceptation immédiate du message dans le journal ou à sa mise en attente. Un message m est ajouté immédiatement dans le journal s'il est linéaire dans J. Il est ajouté à l'ensemble max_J et les messages desquels il

dépend sont retirés de l'ensemble. Son ajout immédiat dans J conduit préalablement à l'ajout de ses dépendances qui sont en attente. Certaines de ses dépendances en attente peuvent être des messages non-linéaires avec un ou plusieurs messages du journal. Ces dépendances non-linéaires ne sont pas ajoutés dans l'ensemble max_J puisqu'il s'agit de dépendances de m. En revanche, un message maximal du journal peut être non-linéaire avec l'une des dépendances du message ajouté. L'ajout d'un message et de ses dépendances en attente peut donc conduire à la présence d'un message dans l'ensemble max_J dont l'auteur est reconnu malintentionné dans J. Pour éviter cette situation, le protocole exécute une opération d'acquiescement avant ces ajouts. L'exécution d'une opération d'acquiescement génère un message d'acquiescement. A l'issue de ces ajouts, l'ensemble max_J contient deux messages : le message d'acquiescement et le message m. Ces deux messages sont linéaires. La livraison d'un message garantit donc l'absence de messages non-linéaire dans l'ensemble max_J.

Lors du processus d'initialisation pour rejoindre la collaboration, un pair vérifie après chaque ajout de message dans le journal l'absence de messages maximaux dont le pair est reconnu malintentionné dans J. □

Démonstration du théorème 4.11. Démontrons d'abord que l'exécution abstraite associée à un journal respecte la propriété C1 de visibilité transitive et la propriété C2 de visibilité non-spéculative. Dans l'exécution abstraite associée au journal, la relation de visibilité et la relation retourne-avant sont construites respectivement à partir de la relation de dépendance et de l'ordre d'ajout. Sans l'opération d'observation implicite, la relation de visibilité et la relation retourne-avant sont donc respectivement un ordre partiel strict et une extension linéaire de la première. L'opération d'observation implicite est défini de manière à être l'élément maximal de ces deux ordres. Nous concluons donc que la relation de visibilité respecte la propriété de visibilité transitive et la propriété de visibilité non-spéculative.

Démontrons que l'exécution abstraite associée à un journal respecte la propriété FJC1 d'ordre linéaire pour chaque pair honnête. Dans le protocole présenté, un pair se considère comme le seul pair honnête. Il nous suffit donc de montrer que la relation de visibilité restreinte aux opérations de ce pair forme un ordre total strict. D'après le Lemme 4.11.3, la relation de dépendance restreinte aux messages du possesseur du journal forme un ordre total strict. L'opération d'observation implicite est l'élément maximal de l'ordre de visibilité. L'ordre de visibilité restreint aux opérations du possesseur du journal est donc un ordre total strict.

Démontrons que l'exécution abstraite associée à un journal respecte la propriété VFJC2. Pour démontrer cette propriété, il nous faut montrer qu'un pair qui suit le protocole ne produit pas de messages et n'accepte pas de messages qui viole cette propriété. Lorsque le pair exécute une opération, il génère un message m qu'il ajoute à son journal. m dépend des messages maximaux max_J du journal. D'après le Lemme 4.11.4, l'ensemble max_J n'inclut pas de messages de pairs reconnus malintentionnés dans J. Le message généré ne déclare donc pas de dépendances dont les pairs sont reconnus malintentionné dans le sous-journal qui inclut ce message et ses dépendances. Lorsqu'un pair vérifie qu'un message est bien-formé, il vérifie notamment que ce message ne déclare pas une dépendance dont

l'auteur est reconnu malintentionné dans le sous-journal qui contient seulement le message et ses dépendances (équation 4.28). Les relations de dépendances directes d'un message correspondent à un sous-ensemble non-strict des dépendances déclarées du message. Un message x_A est immédiatement visible à un message y_A dans l'exécution abstraite A associée au journal tel que x_A et y_A ne sont pas l'opération d'observation implicite si et seulement si le message associé à x_A est une dépendance directe du message associé à y_A. L'exécution abstraite restreinte aux opérations exécutées qui sont associées à des messages du journal respectent donc la propriété VFJC2. Puisque toutes les opérations de l'exécution abstraite sont visibles ou égales à l'opération d'observation implicite, les messages associés aux opérations qui sont immédiatement visibles à l'opération d'observation implicite correspondent à un sous-ensemble des messages maximaux du journal. En suivant le même raisonnement que précédemment, nous en concluons que l'exécution abstraite associée à un journal respecte la propriété VFJC2.

L'exécution abstraite associée à un journal respecte la propriété I1 (opération d'invitation), étant donné que le type de données répliquées inclut cette opération. Démontrons que l'exécution abstraite associée à un journal respecte la propriété I3 (invitation exigée). Lorsqu'un pair vérifie qu'un message m est bien-formé, il vérifie que m dépend directement ou indirectement d'un message qui invite l'auteur de m (équation 4.27 et équation 4.26). Pour s'assurer que ses propres messages dépendent d'un message qui l'invite, un pair s'invite lui-même lorsqu'il initie la collaboration et vérifie que le journal contient une invitation qui l'invite lorsqu'il rejoint une collaboration existante. Puisqu'un message déclare les dépendances de l'ensemble max_J et que max_J correspond aux éléments maximaux du journal, les messages du pair dépendent toujours du message qui l'invite. Par correspondance entre les messages du journal et les opérations de l'exécution abstraite, nous en déduisons que l'invitation d'un pair est visible à toutes les opérations de ce pair.

Démontrons que l'exécution abstraite associée à un journal respecte la propriété I4 (unicité des invitations). L'identifiant d'un pair est composé du *hash* d'un sous-ensemble des méta-données du message qui l'invite. Deux messages bien-formés avec ce même sous-ensemble de méta-données sont identiques. D'après nos hypothéses cryptographiques, l'adversaire de la collaboration ne dispose pas des ressources nécessaires pour produire deux données distinctes avec le même *hash*. Il ne peut donc pas produire deux invitations qui invitent le même pair.

Démontrons que l'exécution abstraite associée à un journal respecte la propriété I2 (opération d'invitation initiale). Lorsqu'un pair initie une collaboration il exécute une première opération qui l'invite lui-même. Le message produit est donc un message d'invitation qui invite son auteur. Le message constitue le premier message ajouté dans le journal. Tout message ultérieur du pair dépend de ce premier message. Lorsqu'un pair rejoint une collaboration ou livre un message, il s'assure que le message est bien-formé. L'une des conditions a respecter est que le premier message du journal soit une invitation qui invite son auteur (équation 4.24). Il vérifie que tout message qui invite son auteur correspond au premier message du journal. Tout autre message doit dépendre d'une invitation qui invite l'auteur du message. Par transitivité, tout message bien-formé dépend du (ou est égal au) premier message du journal. Par correspondance, l'exécution abstraite associée au journal respecte la propriété I2 (opération d'invitation initiale).

Démontrons finalement que l'exécution abstraite associée à un journal respecte la propriété DVFJC2 de rejet d'invitations non-linéaires. Une invitation non-linéaire ne peut pas être acceptée immédiatement dans le journal. Elle est donc mise en attente. De ce fait, le nouvel invité n'est pas inclus dans l'ensemble $\mathsf{invited}_J$ qui contient l'ensemble des pairs invités par des messages du journal. Or tout message dont l'auteur ou dont l'auteur d'une dépendance indirecte ou directe n'est pas un invité connu du journal est mis en attente. Le seul moyen d'ajouter ces messages est d'ajouter d'abord le message d'invitation. Ainsi un message d'un invité connu qui dépend de l'invitation non-linéaire et ne dépend pas des messages dont l'auteur n'est pas un invité connu doit être accepté dans le journal. Par correspondance, l'exécution abstraite associée au journal respecte la propriété DVFJC2. □

4.2.6 Stabilité des messages du journal

Le protocole à journaux tronqués repose sur le concept de stabilité pour tronquer le journal. Il repose sur le protocole à journaux complets. La Definition 4.13 définit ce qu'est un message stable, et le théorème 4.12 créé une correspondance entre messages stables et opérations stables.

Definition 4.13 (Message stable)**.** Dans un journal J, un message m est stable si et seulement si tout message ajouté ultérieurement à J dépend de m.

Théorème 4.12 (Opération et message stables)**.** *Soit un journal J et son exécution abstraite associée A. Si une opération x_A est* DynVFJC*-stable dans A, alors son message associé x_J est stable dans J.*

Démonstration. Soit l'exécution abstraite A' associée à une version future J' de J. A' est une extension de A privée de son opération d'observation implicite $\top_A$. Il est démontrable que si une opération est stable dans A, elle est également stable dans toute extension de A privée de son opération d'observation implicite $\top_A$. Cette démonstration repose sur le fait que le possesseur du journal, et donc l'auteur des opérations d'observation implicites $\top_A$ et $\top_{A'}$ est honnête par définition, et que les opérations visibles à $\top_A$ sont également visibles à $\top_{A'}$. Par conséquent, si x_A est *DynVFJC*-stable, alors son message associé est stable dans J. □

L'exécution abstraite associée à un journal respecte le modèle de cohérence *DynVFJC*. Le modèle de cohérence *DynVFJC* est stabilisable. Toute opération est donc à terme stable. Puisque nous avons une correspondance un-à-un entre les messages du journal et les opérations de son exécution abstraite associée, nous pouvons donc conclure que les messages deviennent à terme stables.

Deux journaux avec le même ensemble de messages, mais possédés par des pairs distincts n'ont pas forcément le même ensemble de messages stables. Cette différence provient du fait qu'ils ont des exécutions abstraites distinctes. Il peut être utile d'introduire une notion de stabilité convergente. Cette utilité est démontrée par le protocole que nous développons dans la section 4.3. Deux journaux avec le même ensemble de messages ont le même ensemble de *messages convergents-stables*.

Definition 4.14 (Message convergent-stable)**.** Soit deux journaux J_1 et J_2 avec le même ensemble de messages, mais avec un possesseur distinct (et éventuellement un ordre d'ajout distinct des messages). Dans un journal J_1, un message m est convergent-stable si et seulement si il est stable dans J_1 et dans J_2.

Théorème 4.13 (Opération stable et message convergent-stable)**.** *Soit un journal J et son exécution abstraite associée A. Si une opération x_A est* DynVFJC*-stable dans l'exécution abstraite A privée de son opération d'observation implicite $\top_A$, alors son message associé x_J est convergent-stable dans J.*

Démonstration. Soit deux journaux J_1 et J_2 avec le même ensemble de messages, mais avec un possesseur distinct (et éventuellement un ordre d'ajout distinct des messages). Les exécutions abstraites associées à J_1 et J_2 privées de leur opération d'observation implicite ont le même ensemble d'opérations et de relations de visibilité. La stabilité *DynVFJC* ne repose pas sur l'ordre d'ajout. Une opération *DynVFJC*-stable dans l'exécution abstraite associée à J_1 privée de son opération d'observation implicite est donc également *DynVFJC*-stable dans l'exécution abstraite associée à J_2 privée de son opération d'observation implicite. D'après le théorème 4.12, le message associé à cette opération est stable dans J_1 et dans J_2. Il est donc également convergent-stable. □

Le protocole présenté permet de produire des journaux dont les messages deviennent à terme stables. Dans la section 4.3, nous étendons ce protocole et nous tirons avantage de la stabilisation des messages pour tronquer le journal.

4.2.7 Cohérence du protocole

Nous nous intéressons désormais aux garanties de cohérence offertes par le protocole. Nous adoptons donc un point de vue global. Nous supposons qu'il existe un oracle qui enregistre l'ensemble des événements qui se produisent sur les pairs honnêtes. Les temps de début et de fin du déroulement des événements sont relevés sur une ligne de temps absolue et infiniment précise.

Chaque exécution d'une opération d'interrogation ou d'une opération de modification par un pair honnête se manifeste par une opération dans l'histoire. Le temps de début et de fin d'exécution permettent de déduire les relations retourne-avant entre les opérations des pairs honnêtes.

Les pairs honnêtes peuvent accepter dans leur journal les messages de pairs malintentionnés. Chaque message d'un pair malintentionné accepté dans le journal d'au moins un pair honnête qui exécute par la suite une opération donne lieu à une opération dans l'histoire de l'exécution. Les pairs honnêtes peuvent accepter à des temps distincts un même message. L'oracle utilise la date à laquelle le message a été accepté pour la première fois de l'exécution dans un journal pour définir les relations retourne-avant entre l'opération x correspondante au message et les autres opérations. Les relations retourne-avant entre cette opération et les opérations des pairs honnêtes sont définies comme suit. Si la date de fin de l'exécution d'une opération est strictement inférieure à la date de première acceptation de x, alors l'opération retourne-avant x. Si la date de début de l'exécution d'une opération est strictement supérieure à la date de première acceptation de x, alors x retourne-avant

cette opération. La définition des relations retourne-avant entre les exécutions des opérations des pairs malintentionnés suit l'ordre de leur date d'acceptation première. Si la date d'acceptation première de x est strictement inférieure à la date d'acceptation première de y, alors x retourne-avant y.

Puisque les événements se déroulent les uns à la suite des autres sur un pair, la restriction de la relation retourne-avant aux messages d'un pair honnête forme un ordre total strict. La propriété BF1 est respectée. L'histoire est donc bien-formée.

Le protocole présenté produit des histoires bien-formées. Nous nous intéressons désormais à montrer que le protocole offre les garanties de cohérences décrites par le modèle de cohérence *DynVFJC*. Pour ce faire, nous montrons que les histoires peuvent être augmentées en des exécutions abstraites bien-formées qui respectent ce modèle de cohérence.

Les exécutions abstraites formées par les histoires auxquelles ont été ajoutées la relation de visibilité comme défini ci-après respectent le modèle de cohérence DynVFJC.

Pour chaque exécution d'une opération de lecture et chaque message du journal au moment de son exécution, une relation de visibilité est créée telle que la destination soit l'opération de lecture et la source soit l'opération de modification à l'origine du message.

Pour chaque exécution d'une opération, une relation de visibilité est créée telle que elle soit la destination et la source soit les précédentes opérations du pair. A noter que certaines relations sont déjà créées par le procédé précédent. Ce procédé permet de créer des relations de visibilité dont la source est une opération de lecture.

Le message d'un pair malintentionné donne lieu à une opération dans l'histoire seulement si un pair honnête accepte ce message dans son journal et qu'il exécute par la suite une opération. Par construction, l'opération d'un pair malintentionné est donc toujours visible à au moins une opération d'un pair honnête. L'exécution abstraite produite est donc bien-formée.

Les relations de visibilité entre les opérations de modification traduisent les relations de dépendances entre les messages qu'elles génèrent. En suivant la même structure de démonstration que celle du théorème 4.11, il est démontrable que l'exécution abstraite respecte le modèle de cohérence *DynVFJC*.

Convergence des pairs honnêtes

Le modèle de cohérence *DynVFJC* considère les pairs honnêtes qui ont été invités dans des embranchements rejetés par les autres pairs honnêtes comme des pairs malintentionnés. En pratique, nous ne pouvons pas faire une telle hypothèse. Nous devons donc admettre que notre protocole peut conduire un pair honnête à diverger des autres pairs honnêtes du groupe. Toutefois, ce dernier peut aisément détecter que le message qui l'a invité est non-linéaire et que ses opérations sont systématiquement rejetés par les autres pairs. Il peut alors demander à un autre pair de l'inviter (sous une nouvelle identité).

Nous démontrons dans le reste de cette sous-section que notre protocole garantit la convergence des pairs honnêtes dont l'invitation ne fait pas partie d'un embranchement rejeté. Ces pairs honnêtes convergent si leurs opérations de modification sont à terme visibles à leurs opérations de lecture.

Lorsqu'un pair honnête exécute une opération de modification, il génère un message qu'il transmet aux autres pairs. L'opération à l'origine du message est visible aux opérations

des autres pairs si ces derniers l'accepte dans leurs journaux. Un pair honnête produit uniquement des messages bien-formés. En revanche, le message d'un pair honnête peut être mis en attente. Pour éviter qu'un message soit mis en attente et le reste indéfiniment, le protocole s'assure que le message d'un pair honnête est toujours accepté immédiatement dans le journal d'un autre pair. Pour ce faire, il s'assure que le message ne dépend pas d'un message dont l'auteur n'est pas connu par les autres pairs. Un pair génère un message d'acquiescement avant d'ajouter le message d'un pair qu'il n'a aps encore reconnu. De ce fait, il force les autres pairs honnêtes à reconnaître le nouveau pair.

4.3 Protocole à journaux tronqués

Le protocole à journaux complets permet de maintenir des journaux View-Fork-Join-Causal Dynamique. Au cours de la collaboration, les messages deviennent à terme stables. Nous exploitons cette observation pour tronquer le journal. Nous présentons dans un premier temps un protocole qui se base sur le protocole à journaux complets. Nous montrons alors qu'une partie du journal peut être tronquée sans menacer la cohérence du contenu répliqué. La transmission d'un journal tronqué à un pair qui rejoint la collaboration ne permet pas à ce dernier de reconstituer une copie du contenu. Nous terminons ainsi par un protocole qui permet d'authentifier un état du contenu avec un journal tronqué.

4.3.1 Structures de données

Lorsqu'un pair ajoute un message dans son journal J, il l'intègre à sa copie S du contenu partagé. S correspond à l'état le plus récent que le pair a du contenu partagé.

Un pair maintient un état I du contenu partagé. Cet état représente un état antérieur de la copie du contenu partagé. Il intègre l'ensemble des messages qui ont été supprimés du journal. Nous détaillons sa construction dans la sous-section suivante. Cet état est accompagné des structures de données dépendantes $\mathsf{knownMalicious}_I$, $\mathsf{invited}_I$, et vv_I. Elles sont mises à jour lorsqu'un message est intégré à l'état I. Leur initialisation et leur mise-à-jour s'effectue de la même manière que les structures de données dépendantes du journal.

Nous parlons d'*empreinte* d'un état I, le *hash* calculé sur I et ses structures de données dépendantes. Le *hash* est calculé de manière à résister aux collisions et aux attaques de préimages. Deux états convergents, c'est-à-dire qui ont intégré le même ensemble de messages et qui ont les même structures de données dépendantes, ont un *hash* identique. L'empreinte de I est notées $\mathsf{footprint}(I)$.

La copie S du contenu partagé est également accompagnée des structures de données dépendantes $\mathsf{knownMalicious}_S$, $\mathsf{invited}_S$, et vv_S. Tout message qui est ajouté au journal J est intégré à la copie S. Nous avons donc les égalités suivantes :

$$\mathsf{knownMalicious}_S = \mathsf{knownMalicious}_J$$
$$\mathsf{invited}_S = \mathsf{invited}_J$$
$$\mathsf{vv}_S = \mathsf{vv}_J$$

4.3.2 Description du protocole

Nous décrivons les changements qui sont apportés au protocole à journaux complets. En plus des événements évoqués dans le précédent protocole, le protocole à journaux tronqués peut également être soumis à la troncature du journal.

Exécution d'une opération d'invitation

L'exécution d'une opération d'invitation génère un message d'invitation. Un message m d'invitation est accompagné des même méta-données qu'un autre message et d'une méta-donnée supplémentaire :

peer_m : l'identifiant de l'auteur de m

num_m : le numéro de m

deps_m : l'ensemble des dépendances déclarées de m

sig_m : la signature cryptographique de m

$\mathsf{footprint}_m$ l'*empreinte* de la copie S du contenu avant l'exécution de l'invitation.

L'empreinte est utilisée pour authentifier un état. Lorsqu'une opération d'invitation est exécutée, le pair dérive un message m d'invitation. Il calcule l'empreinte de la copie S avant l'exécution de l'opération et l'assigne comme empreinte de m.

$$\mathsf{footprint}_m := \mathsf{footprint}(S)$$

Livraison d'un message

La livraison d'un message suit la même procédure que celle présentée dans le protocole à journaux complets. Nous vérifions d'abord si un message est *bien-formé* avant de le traiter afin de l'ajouter immédiatement au journal ou de le mettre en attente. Le traitement d'un message est inchangé. En revanche, nous ajoutons des conditions pour restreindre la définition de ce qu'est un message bien-formé. Ces conditions supplémentaires sont décrites ci-après.

Un message m déclare uniquement des dépendances directes. Cette vérification est nécessaire pour éviter une attaque qui viserait à empêcher la stabilisation du journal. Nous en reparlons lorsque nous développerons les étapes pour tronquer le journal.

$$k, l \in \mathsf{deps}_m \implies \nexists x, y \in W.\, \mathsf{id}(x) = k \wedge \mathsf{id}(y) = l \wedge x \in \mathsf{ctx}_W(y) \tag{4.31}$$

Lorsque le message est une invitation, il vérifie également que l'empreinte est bien-formée. Pour ce faire, il recrée l'état observé par l'auteur de m avant l'exécution de l'opération qui a mené à la production du message m. Dans le protocole à journaux complets, cet état peut être obtenu en intégrant l'ensemble des dépendances de m sur l'état initial du contenu partagé. Cependant, dans le protocole à journaux tronqués, des messages peuvent être supprimés du journal lors de la troncature du journal. Pour remédier à ce problème, le pair maintient un état I qui intègre les messages qu'il a supprimé de son journal J. Nous pouvons donc construire l'état observé par l'auteur de m en intégrant sur I l'ensemble des dépendances de m présentes dans le journal étendu W. Le journal étendu W correspond au journal J auquel est ajouté les messages mis en attente.

Remarque. I ne peut pas intégrer des messages qui ne sont pas des dépendances de m. En effet, seuls des messages stables sont supprimés de J et donc intégrés à I. Par définition si un message est stable, alors tout message ajouté ultérieurement dans le journal étendu J dépend de ce message.

L'état I et ses structures de données dépendantes sont copiées. Ces copies sont temporaires. Chaque message du contexte de m dans W est intégré à l'état temporaire dans l'ordre dans lequel ils ont été ajoutés dans W. Nous obtenons ainsi l'état S_m et ses structures de données dépendantes $\mathsf{invited}_{S_m}$, $\mathsf{knownMalicious}_{S_m}$, et vv_{S_m}. Le pair vérifie alors que l'empreinte de cet état est identique à l'empreinte spécifiée par m.

$$\mathsf{footprint}_m = \mathsf{footprint}(S_m) \tag{4.32}$$

Dans le protocole à journaux complets, nous vérifions qu'un message ne déclare pas une dépendance dont l'auteur est reconnu malintentionné dans le sous-journal qui inclut m et ses dépendances. La suppression de messages du journal peut compromettre le résultat de cette vérification. Certains messages peuvent être perçues comme linéaires, alors qu'ils sont en réalité non-linéaires avec des messages supprimés. Nous devons donc nous assurer que les messages des auteurs des dépendances déclarées de m ne sont pas non-linéaires avec les messages intégrés sur l'état I.

$$\forall \langle p, \_, \_ \rangle \in \mathsf{deps}_m \ \forall x \in W.\, \mathsf{peer}_x = p \implies \mathsf{linear}_I(x) \tag{4.33}$$

Troncature du journal

Les pairs essayent périodiquement de tronquer leur journal pour réduire son occupation mémoire. Ils effectuent cette action indépendamment les uns des autres, sans aucune coordination. Ils peuvent donc effectuer cette action lorsqu'ils sont déconnectés des autres pairs.

Le journal doit être tronqué de sorte à ne pas compromettre la cohérence des copies du contenu partagé. La troncature du journal ne doit pas mener à l'ajout de messages mal-formés au sein du journal. Un nouveau pair qui récupère un journal tronqué doit également être capable de vérifier sa cohérence.

Notre mécanisme de troncature repose sur le concept de stabilité convergente. Pour tronquer son journal, un pair doit d'abord déterminer quels messages sont convergents-stables. Il supprime ensuite un sous-ensemble de ces messages de sorte à permettre à un pair invité de vérifier que le journal a été correctement tronqué. Nous abordons chacune de ces deux étapes.

Messages convergents-stables Pour rejoindre une collaboration, un pair récupère un journal tronqué. Pour vérifier que le journal a été correctement tronqué, le pair et celui qui lui transmet le journal doivent déterminer le même ensemble de messages stables. Pour tronquer leur journal, les pairs déterminent donc les messages convergents-stables. Nous avons précédemment défini ce concept dans le sous-section 4.2.6.

Le protocole à journaux tronqués repose sur le protocole à journaux complets. Un message bien-formé pour le protocole à journaux tronqués est également bien-formé pour le protocole à journaux complets. Les journaux maintenus par le protocole à journaux tronqués sont donc associés à des exécutions abstraites qui respectent le modèle de cohérence *DynVFJC*. Les pairs s'intéressent donc à la stabilité *DynVFJC*.

Puisque les pairs malintentionnés sont capables de générer des messages non-linéaires, la stabilité *DynVFJC* traite différemment les pairs honnêtes et les pairs malintentionnés. Chaque pair suspecte les autres pairs d'être malintentionnés. Lorsqu'un pair détermine la stabilité d'un message, il se considère comme le seul pair honnête. Parce-qu'il est lui-même suspecté malintentionné par les autres pairs, déterminer les messages convergents-stables revient en quelque sorte à considérer tous les pairs, y compris lui-même, comme des pairs malintentionnés.

Un message m est *DynVFJC*-convergent-stable au sein d'un journal J si et seulement si pour chaque énumération de pairs distincts $P \stackrel{\text{def}}{=} \langle p_1, \ldots, p_N \rangle$ de l'ensemble des pairs

invités $\mathsf{invited}_J$, il existe une chaîne de dépendances $m_1 \leq_{\mathsf{dep}} \cdots \leq_{\mathsf{dep}} m_n$ telle que m_1 dépend de m ou est égal à m, et le j-ième pair de la permutation est lié au j-ième message de la chaîne par l'une des relations suivantes : (i) le pair est l'auteur du message (ii) le pair est reconnu malintentionné dans le message (iii) le pair est invité dans le message. S'il existe k tel que le k-ième pair est invité dans le k-ième message, alors il s'agit du dernier message ($k = n$). Autrement la taille de la chaîne est égale à la taille de l'énumération ($n = N$). Nous disons que la chaîne de dépendances est *stabilisante* pour l'énumération P de pairs.

La figure 4.17 montre un exemple d'un journal qui a des messages convergents-stables. Le tableau 4.6 indique le contenu des messages et leurs méta-données. Le tableau 4.7 justifie la stabilité convergente du message b_2 et de ses dépendances. Il indique pour chaque permutation des pairs invités une chaîne de dépendance stabilisante. Par exemple, si nous prenons la permutation $\langle p_M, p_B, p_A \rangle$, alors La châines de dépendances $b_2 \leq_{\mathsf{dep}} b_2 \leq_{\mathsf{dep}} a_5$ est stabilisante pour cette permutation. La chaîne prend source en b_2. p_M est reconnu malintentionné en b_2, p_B est l'auteur de b_2, et p_A est l'auteur de a_5.

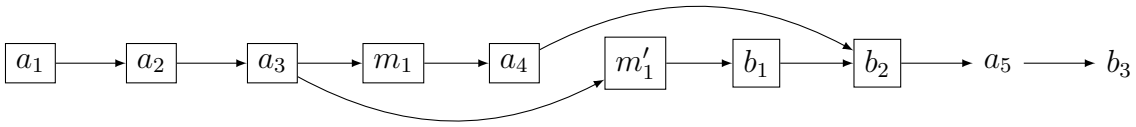


FIGURE 4.17 – Exemple du journal du pair p_A avec des messages convergents-stables. Les messages convergents-stables sont entourés

message m	contenu de m	peer_m	num_m	deps_m
a_1	$\langle \mathsf{invite}, p_A \rangle$	p_A	1	$\emptyset$
a_2	$\langle \mathsf{invite}, p_M \rangle$	p_A	2	$\{a_1\}$
a_3	$\langle \mathsf{invite}, p_B \rangle$	p_A	3	$\{a_2\}$
m_1	$\langle \mathsf{add}, 1 \rangle$	p_M	1	$\{a_3\}$
a_4	$\langle \mathsf{ack} \rangle$	p_A	4	$\{m_1\}$
m'_1	$\langle \mathsf{add}, 2 \rangle$	p_M	1	$\{b_1\}$
b_1	$\langle \mathsf{add}, 3 \rangle$	p_B	1	$\{m'_1\}$
b_2	$\langle \mathsf{add}, 4 \rangle$	p_B	2	$\{a_4, b_1\}$
a_5	$\langle \mathsf{add}, 5 \rangle$	p_A	5	$\{b_2\}$
b_3	$\langle \mathsf{add}, 6 \rangle$	p_B	3	$\{a_5\}$

TABLE 4.6 – Contenu et méta-données des messages du journal de la figure 4.17

Déterminer avec précision la stabilité d'un message tout en passant à l'échelle est un défi difficile. Un algorithme naïf pourrait parcourir chaque permutation et chaque chemin de dépendance du journal. Pour N pairs invités, il y a $N!$ permutations. Et chaque branche dans le graphe démultiplie le nombre de chemins. La complexité d'un tel algorithme est exponentielle.

Nous proposons un mécanisme qui détermine comme convergent-stable un sous ensemble des messages qui le sont. Cet algorithme a une complexité quadratique par rapport

permutation	chaîne de dépendances stabilisante
$\langle p_A, p_B, p_M \rangle$	$a_5 \leq_{\mathsf{dep}} b_3 \leq_{\mathsf{dep}} b_3$
$\langle p_A, p_M, p_B \rangle$	$a_5 \leq_{\mathsf{dep}} a_5 \leq_{\mathsf{dep}} b_2$
$\langle p_B, p_A, p_M \rangle$	$b_2 \leq_{\mathsf{dep}} a_5 \leq_{\mathsf{dep}} a_5$
$\langle p_B, p_M, p_A \rangle$	$b_2 \leq_{\mathsf{dep}} b_2 \leq_{\mathsf{dep}} a_5$
$\langle p_M, p_A, p_B \rangle$	$b_2 \leq_{\mathsf{dep}} a_5 \leq_{\mathsf{dep}} b_3$
$\langle p_M, p_B, p_A \rangle$	$b_2 \leq_{\mathsf{dep}} b_2 \leq_{\mathsf{dep}} a_5$

TABLE 4.7 – Justification de la stabilité convergente du message b_2 et de ses dépendances dans le journal de la figure 4.17

au nombre de pairs invités et au nombre de messages enregistrés dans le journal. Si nous considérons le nombre de pairs invités bien plus petit que le nombre de messages du journal, alors la complexité de cet algorithme est linéaire par rapport au nombre de messages.

Notre algorithme repose sur deux principes :

— Au lieu de considérer chaque permutation des pairs invités, nous considérons une énumération de pairs qui les inclut toutes. Par exemple, l'énumération $\langle p_B, p_A, p_B \rangle$ inclut toutes les permutations de l'ensemble $\{p_A, p_B\}$.

— Au lieu de parcourir chaque chemin de dépendance du journal, nous en parcourons un seul.

Un pair qui récupère un journal tronqué doit être capable de vérifier que le journal est correctement tronqué. Si un pair détermine qu'un message est convergent-stable dans son journal, alors un pair qui récupère ce journal doit également le déterminer convergent-stable. Ils doivent donc parcourir le même chemin dans le journal et considérer la même énumération de pairs.

Le chemin parcouru est déterminé par l'ordre d'ajout des messages dans le journal. Si un message est une dépendance de plusieurs messages, alors le message qui a été ajouté en premier fait partie du chemin parcouru. Si un pair souhaite tester la stabilité du message a_3 dans le journal de la figure 4.17, il parcourt le chemin $a_3 \xrightarrow{\mathsf{dep}} m_1 \xrightarrow{\mathsf{dep}} a_4 \xrightarrow{\mathsf{dep}} b_2 \xrightarrow{\mathsf{dep}} a_5 \xrightarrow{\mathsf{dep}} b_3$.

Nous détaillons ci-après la fonction qui permet de générer une énumération de pairs qui inclut toutes les permutations d'un ensemble ordonné de pairs. Cet algorithme est déterministe. Pour assurer que l'énumération de pairs soit la même quel que soit le pair qui teste la stabilité d'un message, nous devons ordonner l'ensemble des pairs qui est passé en paramètre à cette fonction. Pour ce faire, les identifiants des pairs sont totalement ordonnées. Dans l'exemple que nous considérons, nous supposons l'ordre suivant entre les identifiants des pairs : $p_A < p_B < p_M$.

La formation d'une énumération minimale qui inclut toutes les permutations est un problème ouvert [72]. Toutefois plusieurs articles [73, 74, 75, 76, 77] prouvent des bornes supérieures sur la taille de cette énumération. Pour ce faire, ils décrivent dans leur preuve la génération d'une telle énumération. L'Algorithme 3 adapte la construction mathématique décrite dans l'article de ADLEMAN [73]. Il a une complexité quadratique par rapport à la taille de l'ensemble passé en paramètre. Le tableau 4.8 donne des exemples d'appel.

ensemble ordonné P	$\mathsf{allPermEnum}(P)$
$\langle p_A \rangle$	$\langle p_A \rangle$
$\langle p_A, p_B \rangle$	$\langle p_B, p_A, p_B \rangle$
$\langle p_A, p_B, p_M \rangle$	$\langle p_M, p_A, p_B, p_M, p_A, p_B, p_M \rangle$
$\langle p_A, p_B, p_C, p_M \rangle$	$\langle p_M, p_A, p_B, p_C, p_M, p_A, p_B, p_M, p_C, p_A, p_B, p_M \rangle$

TABLE 4.8 – Énumération de pairs qui inclut toutes les permutations d'un ensemble ordonné de pairs

Algorithme 3 Génération déterministe d'une énumération qui contient toutes les permutations d'un ensemble ordonné de pairs $\langle p_1, \ldots, p_n \rangle$.

```
1: function allPermEnum(⟨p_1, ..., p_n⟩)
2:     if n = 1 then
3:         return ⟨p_1⟩
4:     else if n = 2 then
5:         return ⟨p_2, p_1, p_2⟩
6:     else
7:         let result :=⟨p_n⟩
8:         i := 1
9:         j := 1
10:        while j ≤ n^2 − 3n + 4 do
11:            result := result ++⟨p_{j (mod n)}⟩
12:            if p_{j (mod n)} = p_{n−i} ∧ i ≤ n − 2 then
13:                result := result ++⟨p_n⟩
14:                i := i + 1
15:            end if
16:            j := j + 1
17:        end while
18:        result := result ++⟨p_n⟩
19:        return result
20:    end if
21: end function
```

Ainsi, un message m est convergent-stable si pour l'énumération de pairs $\langle p_1, \ldots \rangle$ généré par l'Algorithme 3 auquel est passé en paramètre l'ensemble ordonné $\mathsf{invited}_J$ des pairs invités, il existe une chaîne $m_1 \leq_{\mathsf{dep}} \ldots \leq_{\mathsf{dep}} m_n$ issue du chemin parcouru qui prend source en m telle que le j-ième pair de l'énumération est lié au j-ième message de la chaîne par l'une des relations suivantes : (i) le pair est l'auteur du message (ii) le pair est reconnu malintentionné dans le message (iii) le pair est invité dans le message. Contrairement à la définition de la stabilité convergente d'un message, nous ne prenons pas en compte la spécificité des invitations. Une invitation est considérée comme une simple observation du pair invité. La chaîne de dépendance doit donc avoir une taille égale à la taille de l'énumération.

Algorithme 4 Détermine si un message m est convergent-stable dans J.

1: **function** convergentStable(m) **require** $x \in J$
2: **let** $Z := \langle\rangle$ ▷ initialisation du journal temporaire
3: **let** $\mathsf{vv}_Z := \mathsf{vv}_I$
4: **let** $\mathsf{knownMalicious}_Z := \mathsf{knownMalicious}_I$
5: **let** $\langle y_1, \ldots \rangle := J$
6: $Z := Z +\!\!+ \langle y_1, \ldots, m \rangle$
7: **let** $\langle p_1, \ldots, p_n \rangle := \begin{cases} \mathsf{allPermEnum}(\mathsf{invited}_J - \mathsf{knownMalicious}_Z) & \textbf{if } \mathsf{max}_Z = \{m\} \\ \mathsf{allPermEnum}(\mathsf{invited}_J) \end{cases}$
8: **let** $j := 1$
9: **let** $\mathsf{prev} := m$
10: $k := \mathsf{length}(Z)$ ▷ débuter le parcours de J en m
11: **while** $j \leq n \wedge k \leq \mathsf{length}(J)$ **do**
12: **if** $y_k = m \vee \mathsf{id}(prev) \in \mathsf{deps}_{y_k}$ **then** ▷ y_k fait-il partie du chemin à parcourir ?
13: **if** $\mathsf{peer}_{y_k} = p_j \vee y_k = \langle \mathsf{invite}, p_j \rangle \vee p_j \in \mathsf{knownMalicious}_Z$ **then**
14: $j := j + 1$ ▷ avance d'un pas dans l'énumération de pairs
15: **else**
16: $\mathsf{prev} := y_k$
17: $Z := Z +\!\!+ \langle y_k \rangle$
18: $k := k + 1$ ▷ avance d'un pas dans le journal J
19: **end if**
20: **else**
21: $Z := Z +\!\!+ \langle y_k \rangle$
22: $k := k + 1$ ▷ avance d'un pas dans le journal J
23: **end if**
24: **end while**
25: **return** $j > n$ ▷ l'énumération de pairs a t-elle été entiérement parcourue ?
26: **end function**

L'Algorithme 4 teste la stabilité du message m passé en paramètre. Il initialise d'abord un journal vide Z. Les structures de données dépendantes vv_Z et $\mathsf{knownMalicious}_Z$ sont initialisées avec l'état des structures de données dépendantes vv_I et $\mathsf{knownMalicious}_I$. Le préfixe minimal du journal J qui inclut m est ajouté au journal temporaire Z. L'énumération de pairs est ensuite générée (ligne 7). L'algorithme introduit une optimisation que nous discutons dans le paragraphe suivant. Les variables j et k permettent respectivement d'itérer sur le journal J et sur l'énumération de pairs. La variable prev enregistre le dernier message traité du chemin à parcourir. Ce chemin prend source en x. Le reste de l'algorithme parcourt l'énumération de pairs et le journal J (lignes 11 à 24). Si le message fait partie du chemin à parcourir (ligne 12), nous avançons d'un pas dans l'énumération de pairs si et seulement si le pair actuel de l'énumération est (i) l'auteur du message actuel, ou s'il est (ii) le pair invité dans le message actuel (le message actuel est une invitation), ou s'il est (iii) reconnu malintentionné dans Z (il est présent dans l'ensemble $\mathsf{knwonMalicious}_Z$). Dès lors que nous n'avançons plus dans l'énumération de pairs, nous progressons d'un pas

dans le journal J. Le message actuel est alors ajouté au journal temporaire Z. Si l'énumération de pairs est entièrement parcourue, le message m est convergent-stable (ligne 25).

L'Algorithme 4 ajoute une optimisation. Puisque la chaîne considérée prend source en m, tous les messages de la chaîne ne peuvent pas dépendre directement d'un message d'un pair reconnu malintentionné en m. Nous pouvons donc considérer une énumération qui contient les permutations des pairs invités auquel est retiré les pairs reconnus malintentionnés en m. Si m est l'unique message des messages maximaux du journal Z, alors m dépend d'un préfixe du journal. Les pairs reconnus malintentionnés en m correspondent donc aux pairs reconnus malintentionnés dans Z. Dans le cas où m ne dépend pas d'un préfixe du journal, nous ne considérons pas cette optimisation.

Pour illustrer cet algorithme, nous testons la stabilité convergente du message m_1 du journal de la figure 4.17. Le message est déterminé convergent-stable par l'algorithme. La valeurs des variables initialisées par l'algorithme entre la lignes 2 et la ligne 13 sont détaillées dans le tableau 4.9. L'évolution de ces variables entre la ligne 14 et 27 est détaillée dans le tableau 4.10.

variable·s	valeur·s initiale·s
m	m_1
Z	$\langle\rangle$
vv_Z	$\emptyset$
$\mathsf{knownMalicious}_Z$	$\emptyset$
$\langle y_1, \ldots, y_{10}\rangle$	$\langle a_1, a_2, a_3, m_1, a_4, m'_1, b_1, b_2, a_5, b_3\rangle$
k	1
$\langle p_1, \ldots, p_7\rangle$	$\langle p_M, p_A, p_B, p_M, p_A, p_B, p_M\rangle$
n	7
j	1
prev	m_1

TABLE 4.9 – Valeurs initiales des variables de l'Algorithme 4 lorsqu'il est appliqué au message m_1 du journal de la figure 4.17.

Suppression de messages convergents-stables Pour éviter de tester la stabilité convergente de plusieurs messages et tirer avantage de l'optimisation qui concerne les messages qui dépendent d'un préfixe du journal, un pair teste périodiquement la stabilité d'un message dont il est l'auteur.

Si le message est convergent-stable, alors le préfixe minimal du journal qui inclut le message est convergent-stable. Le pair ne peut pas simplement supprimer ce préfixe pour tronquer le journal. Il doit conserver suffisamment de messages pour permettre à un pair de vérifier que le journal a été correctement tronqué. Il doit en particulier permettre à un pair de vérifier qu'un message qui ne fait pas partie du préfixe est bien-formé. Pour ce faire, il conserve tous les messages du préfixe qui sont des dépendances de messages qui ne font pas partie du préfixe. Pour simplifier, nous tronquons toujours un préfixe du journal. Une fois le journal tronqué, le pair enregistre l'indice du message déterminé convergent-stable

ligne·s exécutée·s	j	p_j	prev	k	m_k	knownMalicious$_Z$
2 à 10	1	p_M	m_1	1	a_1	$\emptyset$
11, 12, 21, 22	1	p_M	m_1	2	a_2	$\emptyset$
11, 12, 21, 22	1	p_M	m_1	3	a_3	$\emptyset$
11, 12, 21, 22	1	p_M	m_1	4	m_1	$\emptyset$
11 à 14	2	p_A	m_1	4	m_1	$\emptyset$
11 à 13, 16 à 18	2	p_A	m_1	5	a_4	$\emptyset$
11 à 14	3	p_B	m_1	5	a_4	$\emptyset$
11 à 13, 16 à 18	3	p_B	a_4	6	m'_1	$\{p_M\}$
11, 12, 21, 22	3	p_B	a_4	7	b_1	$\{p_M\}$
11, 12, 21, 22	3	p_B	a_4	8	b_2	$\{p_M\}$
11 à 14	4	p_M	a_4	8	b_2	$\{p_M\}$
11 à 14	5	p_A	a_4	8	b_2	$\{p_M\}$
11 à 13, 16 à 18	5	p_A	b_2	9	a_5	$\{p_M\}$
11 à 14	6	p_B	b_2	9	a_5	$\{p_M\}$
11 à 13, 16 à 18	6	p_B	a_5	10	b_3	$\{p_M\}$
11 à 14	7	p_M	a_5	10	b_3	$\{p_M\}$
11 à 14	8		a_5	10	b_3	$\{p_M\}$

TABLE 4.10 – Évolution des valeurs des variables de l'Algorithme 4 lorsqu'il est appliqué au message m_1 du journal de la figure 4.17.

et l'indice du dernier message du journal (celui ajouté le plus récemment) dans le couple stabilityRef. Il s'agit de la référence de stabilité.

Prenons l'exemple du journal de la figure 4.17. p_A le possesseur du journal décide d'essayer de tronquer son journal. Il choisit de tester la stabilité du message a_4. a_4 est déterminé convergent-stable. Le préfixe $a_1 \xrightarrow{\text{dep}} a_2 \xrightarrow{\text{dep}} a_3 \xrightarrow{\text{dep}} m_1 \xrightarrow{\text{dep}} a_4$ est donc convergent-stable. a_3 est une dépendance du message m'_1. Puisque m'_1 ne fait pas partie du préfixe convergent-stable, il ne peut pas supprimer a_3. Le pair supprime donc le préfixe $a_1 \xrightarrow{\text{dep}} a_2$. Sa référence de stabilité correspond au couple $\langle 3, 8\rangle$. Ces deux indices correspondent aux emplacements des messages a_4 et b_3 au sein du journal tronqué. Ces étapes sont décrites dans l'Algorithme 5.

Le pair p_B décide également de tronquer son journal. La figure 4.18 représente son journal. Il choisit de tester la stabilité du message b_2. b_2 est déterminé convergent-stable. Aucun message du préfixe convergent-stable, à l'exception de b_2 n'est une dépendance d'un message qui ne fait pas partie du préfixe. Il supprime donc l'ensemble du préfixe, à l'exception de b_2. Sa référence de stabilité correspond au couple $\langle 1, 3\rangle$. Ces deux indices correspondent aux emplacements des messages b_2 et b_3 au sein du journal tronqué.

Un pair malintentionné pourrait déclarer le premier message du journal complet comme dépendance pour empêcher la troncature des journaux des pairs honnêtes. C'est pourquoi nous limitons au sein de ce protocole les dépendances déclarées aux dépendances directes du message. Un message qui déclare des dépendances indirectes est mal-formé (équation 4.31).

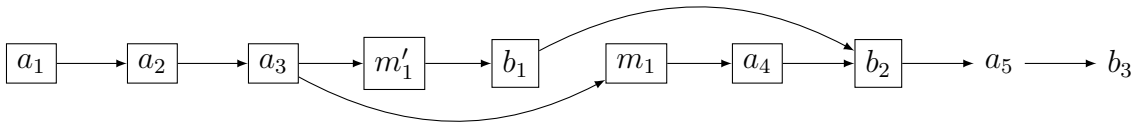


FIGURE 4.18 – Exemple du journal du pair p_B avec des messages convergents-stables. Les messages convergents-stables sont entourés

Algorithme 5 Troncature du journal d'un pair p_i.

```
1:  procedure attemptTruncation_i
2:      let ⟨y_1, ..., y_n⟩ := J
3:      pick k ∈ ℕ* ∧ peer_{y_k} = p_i
4:      if convergentStable(y_k) then
5:          let j := 1
6:          while j < k ∧ ∄l > k. id(y_j) ∈ deps_{y_l} do
7:              J := J − ⟨y_j⟩                          ▷ supprime y_j du journal J
8:              I := integrate_i(I, y_j)                ▷ intégre y_j dans l'état I
9:              j := j + 1                              ▷ passe au message suivant
10:         end while
11:         stabilityRef :=⟨k − j − 1, n − j − 1⟩   ▷ mise à jour des références de stabilité
12:     end if
13: end procedure
```

Initialisation

Un pair peut initier une nouvelle collaboration ou en rejoindre une existante.

Initier une collaboration. Pour initier une nouvelle collaboration, un pair exécute une première opération qui l'invite. L'exécution de cette invitation génère son identifiant de pair.

Rejoindre une collaboration. Pour rejoindre une collaboration, un nouveau pair doit d'abord se faire inviter. Un pair de la collaboration peut inviter à tout moment un nouveau pair. Pour ce faire, il exécute une opération d'invitation qui génère l'identifiant p_i du nouveau pair. Il signe et transmet au nouveau pair un message qui contient l'identifiant p_i, son journal J, son état I, vv_I, $\mathsf{knownMalicious}_I$, et $\mathsf{stabilityRef}$. Un pair malintentionné peut transmettre des données arbitraires.

Le nouveau pair p_i vérifie d'abord l'authenticité du message. Nous notons G le journal reçu pour le distinguer du journal J du pair p_i. Nous n'avons pas besoin de distinguer l'état I de celui reçu, étant donnée que p_i utilise directement l'état I qu'il reçoit.

Nous supposons que le journal reçu G est tronqué. S'il ne l'est pas le pair suit les mêmes étapes que celles présentées dans le protocole à journaux complets. Le nouveau pair p_i construit son journal J et l'état de sa copie S à partir des données reçues. Il initialise les structures dépendantes de son journal à l'aide des structures dépendantes de I et sa copie S est initialisé avec l'état I.

Algorithme 6 Initialisation du journal et de la copie du pair p_i à partir d'un journal tronqué G, d'un état I, des structures de données dépendantes de I, et de la référence de stabilité $\mathsf{stabilityRef}$.

```
 1: procedure init_i(G)
 2:     S := I
 3:     vv_J := vv_I
 4:     knownMalicious_J := knownMalicious_I
 5:     let ⟨l, h⟩ := stabilityRef
 6:     let ⟨y_1, ..., y_n⟩ := G
 7:     check 1 ≤ l ≤ h ≤ n
 8:     J := J ++ ⟨y_1, ..., y_l⟩                ▷ ajout des messages convergents-stables
 9:     check max_J = ⟨y_l⟩
10:     check l = h ⟹ convergentStable(y_l)
11:     let k := l + 1
12:     while k ≤ n do
13:         check wellFormed(y_k)
14:         if accepted_J(y_k) then
15:             let msgs := ctx_pending(y_k) ++ ⟨y_k⟩
16:             if p_i ∈ ackInvited ∧ ∃x ∈ msgs. peer_x ∉ ackInvited then
17:                 execute(ack)                 ▷ exécuter une opération d'acquiescement
18:             end if
19:             J := J ++ msgs
20:             pending := pending − msgs
21:             if ⟨invite, p_i⟩ ∈ msgs then    ▷ p_i est-il invité par l'un des messages ?
22:                 check max_J = {⟨invite, p_i⟩}
23:                 ackInvited := invited_J
24:             end if
25:             check ∀x ∈ max_J. peer_x ∉ knownMalicious_J
26:         else
27:             pending := pending ++ ⟨y_k⟩
28:         end if
29:         check k = h ⟹ convergentStable(y_l)
30:         k := k + 1
31:     end while
32:     check pending = ∅ ∧ p_i ∈ invited_J
33: end procedure
```

L'Algorithme 6 détaille les étapes d'initialisation. Le pair vérifie que la référence de stabilité désigne des messages du journal G et que le premier message apparaît bien avant le second message (ligne 7). Il ajoute à J les messages du journal G qui sont déclarés convergents-stables. Il s'agit des messages qui précédent ou sont égaux au premier message y_l de la référence de stabilité. Il vérifie que y_l dépend d'un préfixe du journal (ligne 9). Pour les messages restants, le pair suit globalement les mêmes étapes que celles présentées pour le protocole à journaux complets (ligne 12 à 32). Une fois que le deuxième message de la référence de stabilité est ajouté au journal, il vérifie que y_l est un message convergent-stable (ligne 10 et ligne 29).

L'ajout d'au moins une invitation permet de vérifier la cohérence de l'état I et donc de la copie S par rapport au journal tronqué. Si un message d'un pair honnête est présent dans le journal tronqué et dépend de l'invitation qui invite p_i, alors p_i est assuré d'avoir une copie et un journal cohérent et authentique.

4.3.3 Cohérence du journal

Le protocole à journaux complets produit des journaux dont les exécutions abstraites associées respectent le modèle de cohérence *DynVFJC*. Nous avons ainsi pu définir des messages convergents-stables dans le journal et proposer le protocole à journaux tronqués qui repose sur la stabilisation des messages du journal.

Dans cette section nous montrons que la troncature du journal ne remet pas en cause sa cohérence. En d'autres termes, le journal tronqué accepte et rejette les mêmes ensembles de messages qu'un journal complet. Il nous suffit de montrer que l'ensemble des messages utilisés pour déterminer si un message est bien-formé et acceptable immédiatement dans le journal sont présents autant dans un journal complet que dans une version tronquée.

Pour tronquer son journal, un pair identifie un message m convergent-stable. Il tronque un préfixe de son journal qui inclut uniquement des messages convergents-stables. Ces messages ne doivent pas être des dépendances de messages qui ne sont pas convergents-stables et doivent être distincts de m. Si les dépendances déclarées d'un message livré ont été préalablement livrées, alors elles sont présentes dans le journal étendu W.

Si les dépendances déclarées de m sont présentes dans le journal étendu W et que l'auteur de m n'est pas reconnu malintentionné, alors il existe au moins un message de l'auteur de m qui n'est pas convergent-stable et est présent dans le journal étendu. Le possesseur du journal peut ainsi vérifier que le numéro de m est contigu au dernier message reçu de l'auteur de m. Si l'auteur de m est reconnu malintentionné, il se peut qu'aucun de ses message ne soit présent dans le journal étendu. Le fait qu'il soit reconnu malintentionné est convergent-stable. Le message sera donc déclaré mal-formé même s'il dépendait correctement d'un de ses messages dont le numéro est contigu à celui de m. Dans le journal complet le message aurait pu être déclaré bien-formé. Cependant, le fait qu'il soit reconnu malintentionné de manière convergente-stable implique que son message ne peut dans tous les cas être ajouté au journal J.

Comme tout autre message du journal, un message d'invitation ne peut être supprimé du journal si les messages qui dépendent de ce message ne deviennent pas convergents-stables. Pour devenir convergent-stable, le pair invité dans l'invitation doit participer à leur stabilisation. Il est donc toujours possible de vérifier que le premier message d'un pair

honnête (numéro égal à 1) dépend de son invitation. En revanche pour un pair reconnu malintentionné, il est possible que cette vérification ne puisse plus l'être. Nous retombons dans le même schéma que celui présenté dans le paragraphe précédent.

Le pair vérifie que l'auteur d'un message et les auteurs des dépendances déclarées de ce message ne sont pas reconnus malintentionnés dans le journal qui regroupe le message et ses dépendances. Un pair est reconnu malintentionné si le journal contient deux messages non-linéaires dont il est l'auteur. La troncature du journal peut conduire à un cas où il existe bien un couple de messages non-linéaires, mais l'un d'entre eux a été supprimé du journal. Or, l'état I intègre l'ensemble des messages supprimés du journal. Ses structures de données dépendantes permettent de tester si certains messages sont non-linéaires avec les messages intégrés dans l'état I. Pour répondre à ce problème, le pair vérifie également qu'il n'existe pas un message non-linéaire avec un message intégré dans l'état I pour l'auteur du message vérifié et les auteurs des dépendances déclarées du message.

Dans les protocoles que nous avons présentés, les pairs honnêtes déclarent les dépendances directes de leurs messages. Les pairs malintentionnés peuvent déclarer des dépendances indirectes. Un pair malintentionné peut ainsi s'assurer qu'un de ses messages déclare une dépendance présente dans le journal d'un premier pair honnête et supprimée d'un journal d'un autre pair honnête. Ce qui conduit à une divergence des deux pairs honnêtes. Pour éviter cette situation, les pairs honnêtes vérifient que les messages déclarent uniquement des dépendances directes. Ainsi le message rejeté par le second pair honnête pour cause d'une dépendance manquante, est rejeté également par le premier pair car la dépendance est reconnue indirecte. Vérifier si les dépendances d'un message sont directes dans un journal tronqué est-il faisable ? Seul un préfixe du journal est tronqué. Ce préfixe contient des messages convergent-stables qui ne sont pas déclarés comme dépendances de messages non-convergent-stables. Étant donné que l'ordre d'ajout des messages est un ordre topologique du graphe de dépendances, il est donc possible de vérifier que chaque dépendance déclarée est directe.

4.3.4 Discussion

Le protocole à journaux tronqués repose sur la présence d'un état I qui intègre les messages supprimés lors de la troncature du journal. Cet état permet de reconstituer des états à partir du journal tronqué et de vérifier ainsi si l'empreinte renseignée dans une invitation est correcte. La conservation de cet état peut être évitée si les types de données répliquées sont équipés d'une opération qui annule l'intégration d'un message. Dans ce cas, l'état observé par le pair qui exécute une invitation peut être reconstitué par l'annulation des messages qui ne sont pas des dépendances de l'invitation.

La stabilisation d'un message requiert la participation de l'ensemble des pairs qui ne sont pas reconnus malintentionnés. Si l'un de ces pairs n'exécute pas d'opérations de modification, il peut bloquer la stabilisation des journaux des autres pairs. Il peut s'agir d'un pair honnête qui n'a pas contribué depuis un long moment ou d'un pair malintentionné qui le fait délibérément. Pour répondre à cette problématique, un pair honnête ou malintentionné devrait être évincé de la collaboration après une certaine durée d'inactivité. Le mécanisme d'éviction pourrait reposer sur le même principe que celui proposé pour les pairs malintentionnés.

4.4 Travaux en relation

4.4.1 Stabilité

Stabilité causale. Baquero et al. [13, 66] introduisent des types de données répliquées qui sont construits sur un journal des messages transmis et acceptés. Les relations de dépendances entre les messages forment ainsi un journal causal. Ils constatent que certaines méta-données d'un message ne sont plus nécessaires une fois que toutes les opérations reçues dépendent de ce dernier. Ils constatent également que certains messages ne sont plus utiles pour répondre aux interrogations soumises au contenu répliqué. Pour déterminer quels messages et quelles méta-données peuvent être supprimés, ils introduisent le concept de stabilité causale au sein de journaux causaux.

Dans ce chapitre, nous avons généralisé la notion de stabilité aux modèles de cohérence. Nous avons ainsi défini la stabilité causale au sein d'exécutions abstraites. Nous avons étendu la stabilité causale à des systèmes dynamiques où de nouveaux pairs peuvent être invités à rejoindre la collaboration. La stabilité causale dynamique peut facilement être adapté aux applications décrites par Baquero et al. [13].

Stabilité et cohérence faible. Golding [78] présente un mécanisme qui permet d'identifier approximativement les messages qui ont été observés par l'ensemble des pairs. Son mécanisme est compatible avec un modèle de cohérence plus faible que le modèle de cohérence causale. En effet, il permet aux pairs de ne pas respecter les causalités entre les exécutions de leurs opérations. Cet affaiblissement permet de borner la taille des méta-données des messages pour les applications qui requiérent par exemple le respect d'un modèle de cohérence *PRAM*[16] [53].

Dans ce manuscrit nous nous sommes limité à des modèles de cohérence qui respectent les causalités entre les opérations exécutées. Leur protocole démontre que la notion de stabilité peut avoir un intérêt dans des systèmes qui respectent des modèles de cohérence plus faible.

Coupure globale. L'utilisation d'une notion de stabilité pour tronquer un journal répliqué n'est pas nouvelle. Sarin et al. [79] introduisent un mécanisme pour supprimer les mises-à-jour qui ne sont plus nécessaires à la résolution de conflits et à la convergence des copies au sein d'une base de données distribuées. Une mise-à-jour peut entrer en conflit avec une mise-à-jour précédemment livrée seulement si cette dernière à un horodatage qui dépasse le horodatage de la première. Une mise-à-jour n'est donc plus nécessaire à la résolution de conflits dès lors que son horodatage est strictement inférieur aux horodatages de l'ensemble des mises-à-jour qui sont ultérieurement livrées. Ils définissent ainsi le concept de *coupure globale*. Un horodatage t est une *coupure globale* dès lors que l'ensemble des mises-à-jour avec un horodatage strictement inférieur à t ont été livrées à l'ensemble des pairs.

16. Il garantit que l'ordre d'intégration des opérations d'un pair correspond à leur ordre d'exécution

La notion de coupure globale s'apparente à celle de stabilité. Toutefois, les horodatages utilisés n'encodent pas de causalité entre les mises-à-jour. Ainsi une mise-à-jour future peut toujours être en concurrence avec une mise-à-jour passée. De plus, la connaissance des mise-à-jours livrées n'est pas suffisante à la détermination d'une *coupure globale*. En effet, les horodatages utilisés par chaque pair ne sont pas garantis à croître de manière monotone. Ils conçoivent ainsi un mécanisme qui permet aux pairs de coopérer de manière asynchrone pour déterminer une *coupure globale*.

Stabilité de livraison de message. BIRMAN et al. [71] introduisent une notion de stabilité dénommée *message stability*. Ils s'intéressent à la stabilité de la livraison d'un message. Une livraison d'un message devient stable dès lors que l'ensemble des destinataires du message ont reçu ce message. Un message concurrent à un message dont la livraison est stable peut toujours être reçu.

4.4.2 Journaux infalsifiables

La litérature propose de nombreux protocoles qui reposent sur un journal infalsifiable [46, 61, 15, 16, 80]. Nous présentons les deux travaux qui se rapprochent le plus des objectifs que nous nous sommes donnés et qui utilisent un modèle de système et un modèle d'adversaire proches des nôtres.

Résumés de journaux. *ASTRO* [61] se base sur un journal infalsifiable des messages transmis et acceptés. Les messages référencent leurs dépendances et sont livrés lorsque les dépendances sont satisfaites. A tout moment, un pair peut décider de réduire la taille de son journal en remplaçant des séquences de messages par un résumé sécurisé. Le résumé référence les dépendances des messages qu'il résume. Lorsqu'un message est produit il référence le résumé sécurisé de l'ensemble des messages précédemment livrés. De ce fait, des séquences de messages peuvent être remplacées par des résumés de manière transparente. Un résumé est construit à l'aide d'un arbre de Merkle [81].

Le protocole présenté permet de garantir le modèle de cohérence Fork-Join-Causal (FJC). Deux pairs honnêtes peuvent en effet résumer des séquences de messages qui diffèrent d'un seul message non-linéaire. Une fois résumé il n'est pas possible d'identifier quels messages sont non-linéaires à moins de récupérer les messages qui sont résumés.

Le calcul d'un résumé requiert un ordre total entre les messages qui est identique quel que soit l'ordre de livraison des messages. Lorsqu'un message est ajouté au journal, le nouvel ordre total n'est pas forcément un suffixe de l'ordre précédent. Il n'est donc pas possible de construire de manière incrémentale un résumé pour vérifier les messages livrés ultérieurement et pour construire les prochains messages. Ce qui induit un surcoût non-négligeable. L'article ne discute pas de ce problème. De même, l'article ne discute pas des messages qui dépendent d'un message inclus dans un résumé. Si le pair ne possède pas les messages résumés, il ne peut pas accepter ces messages.

Notre protocole garantit un modèle de cohérence plus fort qui évince les pairs malintentionnés. Il garantit également la convergence des pairs honnêtes.

État authentifié Au moment de la rédaction de ce manuscrit, KOLLMANN et al. [80] ont présenté *SnapDoc*. *SnapDoc* se place dans un contexte similaire au nôtre et a pour objectif de masquer l'historique de la collaboration aux futur·e·s collaborateur·ice·s. L'historique de la collaboration correspond à un journal infalsifiable dans lequel sont enregistrés les messages échangés par les pairs. Les pairs ne transmettent pas leur journal aux nouveaux pairs. Pour rejoindre la collaboration, un nouveau pair récupère l'état d'une copie du contenu partagé.

SnapDoc tolère la présence de pairs malintentionnés. Un pair malintentionné peut falsifier un état du contenu, avant de le transmettre à un nouveau pair. Pour limiter les falsifications, l'état est accompagné de plusieurs données infalsifiables qui permettent son authentification. Il est en particulier accompagné du dernier message de chaque pair de la collaboration et de preuves qui démontrent l'absence d'embranchements (de messages non-linéaires) au sein du journal.

Le mécanisme d'authentification fait l'hypothèse que l'état de la copie du contenu partagé est décomposable en éléments irréductibles. Chaque message inclut un accumulateur cryptographique et un *hash mh*. L'accumulateur cryptographique est calculé sur l'ensemble des éléments irréductibles de l'état au moment de la génération du message. Le *hash mh* est calculé sur l'ensemble des arbres de *Merkle* [81] maintenus par l'auteur du message. Chaque pair maintient un arbre de *Merkle* pour chaque pair (y compris lui-même) dans lequel il enregistre les messages qu'il reçoit de ce dernier. Un nouveau pair vérifie que pour chaque message m qui accompagne l'état s, les éléments irréductibles de s observés par l'auteur de m sont bien présents dans l'accumulateur de m. Il vérifie également qu'il dispose d'une preuve que le *hash mh* de m couvre bien des préfixes des arbres de *Merkle* du pair qui lui a transmis l'état.

L'utilisation d'accumulateurs cryptographiques a un surcoût important qui rend le protocole inadapté à la réplication de contenus composés d'un nombre important d'éléments irréductibles. En effet, l'accumulateur ne peut pas être calculé de manière incrémentale lorsque des éléments irréductibles sont supprimés de la décomposition de l'état. De plus, chaque pair doit vérifier que l'accumulateur cryptographique d'un message est correctement calculé, ce qui implique de reconstituer l'état observé par l'auteur du message. Les auteurs de ce protocoles préconisent son utilisation pour des contenus de petites tailles qui sont soumis à peu de modifications.

Notre protocole à journaux tronqués et *SnapDoc* se différencient d'abord sur l'objectif qu'ils visent. Notre protocole à journaux tronqués n'a pas pour objectif de masquer l'historique de la collaboration et *SnapDoc* n'a pas pour objectif de réduire l'occupation mémoire du journal infalsifiable. Toutefois, chacun remplie en partie l'objectif de l'autre. Notre protocole permet la troncature du journal qui masque donc en partie l'historique de la collaboration aux nouveaux pairs. *SnapDoc* permet à un nouveau pair de débuter la collaboration sans l'historique de la collaboration. Le journal d'un nouveau pair occupera donc moins de mémoire que le journal d'un pair existant.

Notre protocole à journaux tronqués et *SnapDoc* se différencient également sur les garanties de cohérence qu'ils offrent. *SnapDoc* autorise l'apparition d'embranchements, mais ne permet pas leurs jonctions. Les journaux et les copies des pairs honnêtes peuvent donc diverger. Notre protocole respecte un modèle de cohérence plus faible qui permet la jonction des embranchements et l'éviction à terme des pairs malintentionnés. Il permet

la convergence des copies des pairs honnêtes à la condition que l'invitation de chaque nouveau pair honnête soit acceptée par un pair honnête existant

Du point de vue de la sécurité, *SnapDoc* offre une garantie supplémentaire par rapport à notre protocole. Il ne permet pas à un pair malintentionné d'envoyer un état falsifié dans lequel se trouve des éléments irréductibles produits par un pair honnête. Il permet en effet l'authentification de chaque éléments irréductibles qui compose l'état. Cependant, le pair malintentionné à la liberté de supprimer des éléments irréductibles d'un état initialement authentique sans que le nouveau pair puisse le détecter. Cette faculté provient du fait que la composition de l'accumulateur cryptographique est inconnue. Similairement à notre protocole, un nouveau pair est certain qu'il a un état authentique seulement lorsqu'il reçoit le message d'un pair honnête qui dépend de l'invitation du nouveau pair.

Du point de vue de l'utilisabilité, *SnapDoc* ne supporte pas tous les type de données répliquées. Il suppose l'utilisation de *TreeDoc*[17] car les éléments irréductibles que l'intégration d'un message ajoute à la décomposition de l'état de la copie sont déductibles du message. Il s'agit d'une contrainte très spécifique qui n'est pas toujours rencontrée. Par ailleurs, un nouveau pair peut uniquement accepter des messages qui dépendent de son invitation. Il rejette les messages qui sont concurrents à son invitation. Pour remédier à ce problème, l'article propose la récupération et l'authentification d'un nouvel état qui couvre les messages concurrents.

Chaque protocole pourrait être amélioré en empruntant des mécanismes de l'autre protocole. *SnapDoc* pourrait par exemple utiliser le concept de stabilité pour tronquer le journal d'un pair. Notre protocole pourrait reposer sur des accumulateurs cryptographiques pour améliorer l'authentification de certains types de contenu.

4.5 Conclusion

Les journaux infalsifiables permettent de sécuriser la convergence d'un contenu partagé en présence de pairs malintentionnés. La conservation de l'ensemble du journal a un coût. Pour remédier à ce problème, nous avons proposé un protocole qui permet de tronquer le journal. Nous avons également proposé un protocole pour authentifier un état à partir d'un journal tronqué. Un pair peut ainsi récupérer un journal tronqué et un état pour rejoindre une collaboration.

La troncature du journal repose sur le concept de stabilité. Nous avons exploré ce concept avec le modèle de cohérence causale et le modèle de cohérence VFJC. Nous avons également introduit deux nouveaux modèles de cohérences : le modèle de cohérence causale dynamique et le modèle de cohérence VFJC dynamique. Ces deux modèles prennent en compte l'aspect dynamique des groupes de collaboration.

17. Ce type de données répliquées est présenté dans le chapitre 5.

Chapitre 5

Séquences répliquées synchronisées par différences

Sommaire

Une séquence est une abstraction utile à la conception d'applications de collaboration telles que des éditeurs collaboratifs de texte [33, 82]. *EtherPad*[18], *Google Docs*[19], et *ShareLaTeX*[20] sont des exemples d'éditeurs collaboratifs de texte. Ils ont répandu l'usage de l'édition collaborative au sein des communautés scientifiques, et au sein des organisations économiques, sociales, et politiques.

Ces éditeurs collaboratifs reposent sur des infrastructures centralisées. Les collaborateur·ice·s sont connecté·e·s à un serveur. Toute modification qu'elles et ils effectuent sur le document textuel est transmise au serveur. Le serveur transmet alors une mise-à-jour à l'ensemble des collaborateur·ice·s. Cette architecture engendre des problèmes de confidentialité, de vie privée, de censure, et de sécurité. Elle souffre également d'un problème de disponibilité, de latences élevées, d'un passage à l'échelle pauvre, et d'une restriction des formes que peut prendre une collaboration.

18. `https://etherpad.org`
19. `https://docs.google.com`
20. aujourd'hui fusionne avec Overleaf : `https://www.overleaf.com`

Afin de répondre à ces problématiques, de nouveaux travaux proposent de fonder les éditeurs collaboratifs sur des protocoles de réplication optimiste de séquences de caractères [21, 83, 22, 24]. Chaque collaborateur·ice dispose d'une copie du document sur laquelle il ou elle exécute ses opérations. L'exécution des opérations génère des messages qui sont intégrés à la copie. Les messages sont transmis aux autres collaborateur·ice·s qui les intègrent à leur tour.

Les séquences répliquées proposées dans la littérature sont synchronisées par opérations. Lorsqu'un pair exécute une opération sur sa copie, il dérive un message qui encapsule une opération intégrable sur sa copie et les copies des autres pairs. Une opération intégrable dépend d'un ensemble d'opérations intégrables. Par exemple, la suppression d'une valeur doit être intégrée après son insertion. La détection de ces dépendances peut être complexe. Pour simplifier, les protocoles de réplication de séquences supposent généralement une livraison causale des opérations intégrables.

La livraison causale des opérations introduit des latences dans l'ensemble du système. Il suffit qu'une opération ne soit pas livrée pour empêcher la livraison des opérations intégrables qui dépendent d'elle. Ces latences peuvent compromettre des sessions de collaboration en simultané en augmentant le risque d'apparition de conflits de modification. Les collaborateur·ice·s peuvent adopter des stratégies pour limiter les conflits et ne plus tirer pleinement avantage de l'édition collaborative en simultané [5]. Ces problèmes limitent également le passage à l'échelle à des sessions de collaborations qui incluent des centaines de collaborateur·ice·s. La livraison causale des opérations limite la conception de protocoles de synchronisation qui pourraient tirer avantage d'un modèle de livraison aux hypothèses plus faibles.

La synchronisation par états résout ces problèmes. Elle est particulièrement adaptée à des types de données répliquées simples tels que les compteurs [18]. Elle consiste à transmettre directement l'état de la copie aux autres pairs. Les pairs fusionnent les états qu'ils reçoivent à leur copie. Des états peuvent être omis, dupliqués, et fusionnés dans un ordre quelconque. La résilience de ce modèle de synchronisation réduit considérablement les hypothèses de livraison des messages : la livraison des états peut être non-fiable et non ordonnée. La synchronisation par états présente toutefois un surcoût trop important pour des types de données répliquées plus élaborés tels que les ensembles ou les séquences. Au fur et à mesure que l'état croît, le coût de communication et de fusion s'accroît [70].

La synchronisation par différences d'états [18] a montré qu'il était possible de conserver les avantages de la synchronisation par états tout en réduisant leur coût en communication et en fusion [70]. Lorsqu'un pair exécute une opération, il génère un message qui contient une différence d'états. Les protocoles sont conçus de manière à ce que cette différence soit bien plus petite que l'état complet. Les différences d'états doivent être transmises au moins une fois. Les hypothèses de livraison des messages restent faibles. Ce qui ouvre à un large espace de conception de protocole de livraison de messages.

Ces observations nous conduisent à formuler la question de recherche suivante : un protocole synchronisé par différences d'états pour l'édition collaborative en simultané de texte existe-t-il ? Dans un premier temps nous présentons un état de l'art sur les séquences répliquées. Dans un second temps nous présentons un protocole de réplication de séquences par différences d'états. Ce protocole est présenté de manière générique à l'aide d'un formalisme que nous introduisons.

5.1 État de l'art

Dans cette section, nous présentons le type abstrait de données répliquées Séquence, ainsi que les séquences répliquées proposées dans la littérature. Pour finir, nous présentons une classification des séquences répliquées proposées dans la littérature.

5.1.1 Séquences répliquées sans conflits

Une séquence arrange de manière ordonnée des valeurs. La composition de la séquence évolue au cours du temps : des valeurs peuvent être insérées et supprimées de la séquence. Pour ce faire, le type abstrait Séquence dispose d'une opération d'insertion ins et d'une opération de suppression del. Il dispose également d'une opération de lecture rd. L'exécution de l'opération $\mathsf{ins}(n, v)$ insère la valeur v avant la n-ième valeur de la séquence. L'indexage des valeurs commence à 0 et la valeur v est prise dans un ensemble V. Ainsi, l'insertion d'une valeur en $n = 0$ revient à insérer la valeur en début de séquence. Lorsque n est supérieur ou égal à la taille de la séquence, la valeur est insérée en fin de séquence. L'exécution de l'opération $\mathsf{del}(n)$ supprime la n-ième valeur. Si n ne correspond pas à l'indice d'une valeur, la dernière valeur est supprimée. Une lecture rd récupère le n-uplet de valeurs qui représente la séquence. Si les valeurs sont des caractères, ce n-uplet est une chaîne de caractères. $V^{n\in\mathbb{N}_0}$ correspond à l'ensemble des n-uplets de valeurs prises dans V. La signature syntaxique du type abstrait Séquence est ainsi spécifiée comme suit :

$$\mathsf{Op}_{\mathsf{Seq}\langle V\rangle} \stackrel{\text{def}}{=} \{\mathsf{ins}(n, v), \mathsf{del}(n), \mathsf{rd} \mid v \in V \wedge n \in \mathbb{N}_0\}$$
$$\mathsf{Val}_{\mathsf{Seq}\langle V\rangle} \stackrel{\text{def}}{=} V^{n\in\mathbb{N}_0}$$

Pour simplifier les illustrations, nous introduisons l'opération $\mathsf{ins}(n, \langle v_1, \ldots, v_k\rangle)$ qui ajoute le n-uplet de valeurs $\langle v_1, \ldots, v_k\rangle$ avant la n-ième valeur. Cette opération revient à exécuter k opérations d'insertion qui se succèdent $o_1 \downarrow \cdots \downarrow o_k$ avec $\mathsf{call}(o_i) = \mathsf{ins}(n + i - 1, v_i)$ et $1 \leq i \leq k$. Nous employons cette opération uniquement dans les illustrations. La figure 5.1 donne un exemple d'utilisation de ces opérations et suit la spécification usuelle d'une séquence.

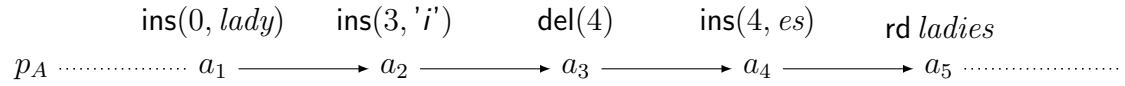


FIGURE 5.1 – Séquence répliquée de caractères modifiée et interrogée par un seul pair. La séquence suit la spécification habituelle d'une séquence non-répliquée.

L'utilisation d'une séquence classique n'est pas adaptée à l'édition collaborative. La réplication optimiste de séquences permet à plusieurs pairs de modifier et d'interroger à tout moment une même séquence. Chaque pair possède une copie de la séquence sur laquelle il exécute immédiatement ses opérations de modification et d'interrogation. Dans la sous-section 3.3.1 nous avons montré que les opérations non-commutatives conduisent à des conflits d'intention si leurs effets usuels sont intégrés sans traitement particulier. Il s'agit d'opérations incompatibles. Les opérations de modifications d'une séquence reposent

sur des indices d'insertion et de suppression. L'indice associé à une valeur est susceptible de changer après l'exécution d'une opération. Lorsqu'une valeur est insérée (respectivement supprimée) avant la n-ième valeur, cette dernière devient la $n+1$-ième valeur (respectivement la $n-1$-ième valeur). Il en résulte que la majorité des opérations sont incompatibles. Seules deux suppressions qui affectent le même indice sont compatibles. Dans la lignée des travaux sur la réplication optimiste de séquences [21, 84], nous soutenons qu'un pair ne souhaite pas insérer une valeur avant la n-ième valeur ou supprimer la n-ième valeur : un pair souhaite insérer une valeur entre deux autres valeurs et supprimer une valeur bien déterminée. Dans la figure 5.1, l'intention de l'opération a_2 est d'insérer '*i*' entre '*d*' et '*y*'.

Dans une séquence, il est toujours possible d'insérer une valeur quelconque entre deux autres valeurs. L'ordre formé entre les valeurs est donc dense. Dans une séquence répliquée deux valeurs peuvent être insérées en parallèle entre un même couple de valeurs. Les deux valeurs insérées en parallèle ne sont pas ordonnées entre-elles. L'ordre entre les valeurs est donc partiel. La séquence est une extension linéaire de cet ordre partiel. Pour obtenir des copies convergentes de la séquence partagée, cette extension linéaire doit être déterminée par conception.

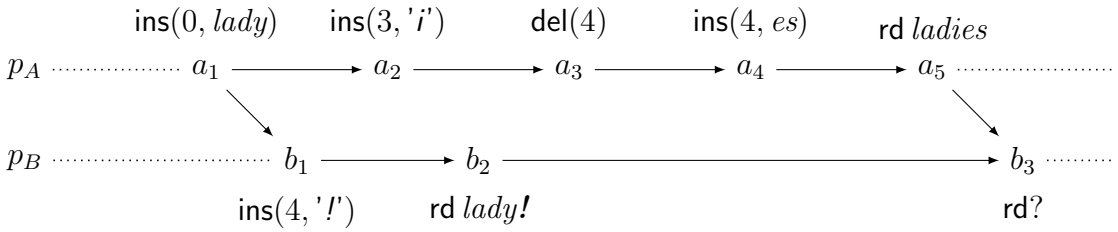
Une spécification complète d'une séquence répliquée détermine l'ordre total entre les valeurs. Elle spécifie notamment l'ordre entre deux valeurs insérées en parallèle au même emplacement. Elle peut par exemple les ordonner en fonction des dates auxquelles elles ont été insérées, en fonction des pairs qui les ont insérées, ou toutes autres caractéristiques qui les différencient. Ce choix semble arbitraire et difficile à justifier. Il conduit à de nombreuses spécifications pour les séquences répliquées. D'un point de vue abstrait, il y a peu d'intérêt de s'intéresser à des spécifications complètes. C'est pourquoi, la littérature présente des spécifications partielles qui se contentent de spécifier un ordre partiel entre les valeurs et de définir la séquence comme une extension linéaire de cet ordre partiel. Une spécification complète devient alors un raffinement d'une spécification partielle qui choisit une linéarisation de l'ordre partiel. La littérature présente trois spécifications partielles :

La spécification faible [85] ne prend pas en compte les valeurs supprimées dans l'ordre partiel entre les valeurs. Dans l'exécution abstraite de la figure 5.2a, si nous ne prenons pas en compte la valeur supprimée '*y*', alors p_A insère la chaîne '*ies*' entre '*d*' et la fin de la séquence. De même, p_B insère '*!*' entre '*d*' et la fin de la séquence. Nous obtenons ainsi l'ordre partiel illustré dans la figure 5.2b. Cet ordre admet 4 extensions linéaires. Le protocole de réplication utilisée par *Google Docs* respecte la spécification faible [86].

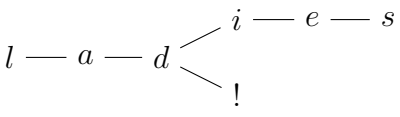
La spécification forte [85] prend en compte l'ordre partiel entre toutes les valeurs, y compris les valeurs supprimées. Dans l'exécution abstraite de la figure 5.2a, si nous prenons en compte la valeur supprimée '*y*', alors p_A insère '*i*' entre '*d*' et '*y*'. Il insère également la chaîne '*es*' entre '*y*' et la fin de la séquence. p_B insère '*!*' entre '*y*' et la fin de la séquence. Nous obtenons ainsi l'ordre partiel illustré dans la figure 5.2c. Cet ordre admet 3 extensions linéaires.

La spécification forte sans entrelacement [87, 88] évite les entrelacements de n-uplet de valeurs insérés en parallèle. Dans l'exécution abstraite de la figure 5.2a, la chaîne '*es*' est insérée en parallèle du caractère '*!*'. La spécification forte sans entrelacement produit le même ordre partiel que la spécification forte, mais elle rejette l'extension linéaire qui produit un entrelacement, c'est-à-dire la chaîne *ladie!s*. Le caractère '*!*' apparaît nécessairement avant ou après la chaîne '*es*'.

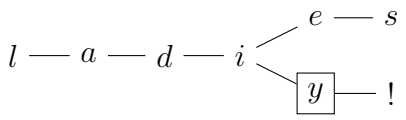
Pour respecter une spécification, les protocoles de réplication optimiste proposés dans la littérature mettent en œuvre différentes stratégies. Dans ce manuscrit nous nous intéressons particulièrement aux CRDTs. Un CRDT encapsule le protocole de réplication optimiste derrière une interface familière. Dans le cas d'une séquence répliquée, l'utilisateur·ice modifie la séquence et la lit de la même manière qu'il ou elle le ferait avec une séquence classique. Le protocole se charge d'exécuter les opérations et de synchroniser la copie de la séquence partagée avec les autres copies de la séquence.



(a)



(b)



(c)

rval(b_3)	Spéc. faible	Spéc. forte	Spéc. forte sans entrelacement
ladies!	✓	✓	✓
ladi!es	✓	✓	✓
ladie!s	✓	✓	
lad!ies	✓		

(d)

FIGURE 5.2 – Illustration des différentes spécifications partielles d'une séquence répliquée. (a) L'exécution abstraite considérée respecte le modèle de cohérence causale. (d) Les valeurs de retour admises de l'opération de lecture b_3 dépendent de la spécification choisie. Elles correspondent à des extensions linéaires des ordres partiels induits par la spécification faible (b) ou la spécification forte (c).

5.1.2 Séquences répliquées de la littérature

Plusieurs CRDTs séquences sont proposés dans la littérature. Un CRDT séquence peut exécuter une opération d'insertion, une opération de suppression, et une opération de lecture. L'exécution d'une opération d'insertion ou d'une opération de suppression génère un message. Le message est intégré à la copie de la séquence avant d'être transmis aux autres pairs qui l'intègrent à terme à leur copie. Nous détaillons chacune de ses procédures pour chacun des CRDTs que nous présentons. Nous caractérisons également la structure du CRDT et des messages qu'il génère.

WOOT [21] est le premier CRDT séquence présenté dans la littérature. Plusieurs optimisations de *WOOT* [89, 20] ont été proposées. Il prend à la lettre la formulation de l'intention d'une insertion : une valeur référence les deux valeurs entre lesquelles elle est insérée.

WOOT référence les valeurs à l'aide de leur identifiant. L'identifiant d'une valeur est constitué par l'identifiant du pair qui l'a inséré et un nombre séquentiel. Chaque pair maintient son propre nombre séquentiel qu'il incrémente avant chaque exécution d'une opération d'insertion. Ce nombre correspond donc au nombre de valeurs insérées par le pair. Si v est la n-ième valeur insérée par le pair p_i elle est associée à l'identifiant $\langle p_i, n\rangle$. Deux identifiants sont réservés pour identifier le début et la fin de la séquence.

Les identifiants des valeurs sont également utilisés pour déterminer l'ordre entre deux valeurs insérées en parallèle au même emplacement. L'algorithme de placement de valeurs est détaillé ci-après. Il se base sur un ordre lexicographique entre les identifiants des valeurs et un ordre total entre les identifiants des pairs. Un identifiant $\langle p_i, n\rangle$ est supérieur à un identifiant $\langle p_j, m\rangle$ si et seulement si p_i est supérieur à p_j ou si p_i est égal à p_j et n est supérieur à m.

Une séquence *WOOT* est une liste des valeurs insérées auxquelles sont associées des méta-données. Les méta-données d'une valeur incluent l'identifiant de la valeur, et les identifiants des valeurs entre lesquelles elle a été initialement insérée.

Lorsqu'un pair exécute une opération d'insertion de la valeur v avant la n-ième valeur, il procède en plusieurs étapes. Il génère d'abord un nouvel identifiant pour v et récupère les identifiants de la $n-1$-ième valeur et de la n-ième valeur. Il s'agit des valeurs entre lesquelles la valeur doit être insérée. Le pair génère alors un message d'insertion qui contient la valeur, l'identifiant de la valeur, et les identifiants des valeurs entre lesquelles la valeur doit être insérée. Le message est intégré à la copie de la séquence et est transmis aux autres pairs.

L'intégration d'un message d'insertion d'une valeur v se déroule comme suit. Le message contient les identifiants des valeurs v_1 et v_2 entre lesquelles v doit être insérée. Ces deux valeurs sont récupérées. Des valeurs peuvent être présentes entre v_1 et v_2. Il s'agit de valeurs insérées en parallèle de v. Pour déterminer où placer v, le pair applique l'algorithme récursif suivant. Il récupère l'ensemble des valeurs comprises entre v_1 et v_2. Si l'ensemble est vide, alors la valeur est simplement placée entre v_1 et v_2. Dans le cas contraire, il récupère le sous-ensemble des valeurs qui référence des valeurs qui ne sont pas comprises entre v_1 et v_2. Ces valeurs ont été insérées en parallèle de v_1 et v_2. L'emplacement relatif de v par rapport à ces dernières est déterminé à l'aide de l'ordre lexicographique entre leurs identifiants.

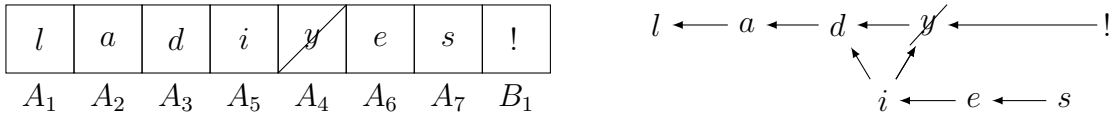


FIGURE 5.3 – Exemple d'une séquence *WOOT*. (a) Séquence de valeurs et pierres tombales associées à leurs identifiants. Les pierres tombales sont représentées par un caractère barré. (b) Valeurs référencées par les valeurs de la séquence. Les références vers le début et la fin de la séquence sont omises.

L'algorithme est répété en considérant les deux valeurs entre lesquelles v doit être placée comme les valeurs v_1 et v_2.

Pour illustrer cet algorithme, nous reprenons l'exemple de la figure 5.2a. Une fois l'ensemble des modifications intégrées, la séquence *WOOT* de la figure 5.3 est obtenue. Un identifiant $\langle p_i, n\rangle$ est noté i_n. p_B intègre les messages d'insertion générés par l'exécution des opérations a_2, a_3, et a_4 avant d'exécuter l'opération de lecture b_3. p_B débute avec la séquence '*lady!*'. Le caractère '*i*' référence les caractères '*d*' et '*y*'. Aucun caractère n'est déjà présent entre ces deux caractères. '*i*' est donc simplement placé entre eux. p_B obtient la séquence '*ladiy!*'. Il intègre ensuite la suppression du caractère '*y*'. Le caractère '*e*' référence le '*i*' et la fin de la séquence. Or il existe un ensemble de caractères entre '*i*' et la fin de la séquence. Nous devons déterminer l'ordre entre '*e*' et ces caractères. Cet ensemble correspond à la pierre tombale '*y*' et au caractère '*!*'. Seul '*y*' référence des valeurs qui ne sont pas comprises entre les valeurs référencées par '*e*'. L'identifiant de '*y*' est plus petit que l'identifiant de '*e*' ($A_4 < A_6$). '*e*' doit donc être placé entre cette pierre tombale et la fin de la séquence. Or '*!*' est entre ces deux bornes. L'identifiant de '*e*' est plus petit que l'identifiant de '*!*' ($A_6 < B_1$), il est donc placé avant. En répétant l'algorithme pour le caractère '*s*', on en déduit que '*s*' est placé entre '*e*' et '*!*'.

Lorsqu'un pair exécute une opération de suppression il détermine d'abord l'identifiant de la valeur à supprimer. Il génère un message de suppression qui contient cet identifiant. Ce message est intégré à la copie de la séquence et est transmis aux autres pairs.

L'intégration d'un message de suppression est relativement simple. L'identifiant de la valeur à supprimer est récupéré du message. Une valeur ne peut pas être définitivement supprimée, étant donné que d'autres valeurs sont susceptibles de la référencer. La valeur est donc remplacée par une pierre tombale. Les méta-données associées à la valeur sont conservées. Elles sont associées à la pierre tombale qui lui fait place.

L'exécution d'une opération de lecture renvoie la séquence des valeurs sans les pierres tombales. La conservation des valeurs supprimées sous forme de pierres tombales fait croître la séquence de manière monotone. Cette croissance réduit l'efficacité de l'exécution des opérations et de l'intégration des messages.

WOOT repose sur une synchronisation par opérations. Les messages doivent être livrés exactement une fois et dans un ordre spécifique. Un message de suppression doit être livré après le message qui insère la valeur à supprimer. Pour insérer une valeur, les valeurs qu'elle référencent doivent être déjà présentes dans la séquence. Un message d'insertion doit donc être livré après les messages d'insertion des valeurs référencées.

Replicated Growable Array (RGA) [24] améliore l'approche proposée par *WOOT*. Une valeur référence uniquement la valeur qui la précède dans la séquence. Deux valeurs qui référencent la même valeur sont ordonnées entre elles à l'aide de leurs identifiants respectifs.

L'identifiant d'une valeur est formé d'un horodatage et de l'identifiant du pair. L'horodatage est produit à l'aide d'une horloge de *Lamport* [55]. L'identifiant associé a une nouvelle valeur est donc toujours supérieur à l'horodatage de l'ensemble des valeurs et pierres tombales déjà présentes dans la copie de la séquence.

L'ordre entre deux valeurs qui référencent la même valeur est déterminé par un ordre lexicographique entre leurs identifiants et un ordre total entre les identifiants des pairs. Un identifiant $\langle n, p_i \rangle$ est supérieur à un identifiant $\langle m, p_j \rangle$ si et seulement si l'horodatage n est supérieur à l'horodatage m ou si n est égal à m et l'identifiant du pair p_i est supérieur à l'identifiant du pair p_j.

Une séquence *RGA* est une liste de valeurs auxquelles sont associées leur identifiant. Contrairement à *WOOT*, la séquence n'a pas besoin de retenir les références entre les valeurs. Cette caractéristique résulte d'un algorithme de placement des valeurs plus simple.

Lorsqu'un pair exécute une opération d'insertion de la valeur v avant la n-ième valeur de la séquence, il génère un message d'insertion. Le message d'insertion contient la valeur, l'identifiant de la valeur, et l'identifiant de la $n-1$-ième valeur. La $n-1$-ième valeur correspond à la valeur après laquelle v doit être insérée. Par la suite nous constaterons que les identifiants de v et de la n-ième valeur conduisent nécessairement à placer v avant la n-ième valeur. L'intention d'insérer v entre la $n-1$-ième valeur et la n-ième valeur est donc respectée à la fois par le référence à la $n-1$-ième valeur et par le choix des identifiants associés aux valeurs. Le message est intégré à la copie de la séquence et est propagé aux autres pairs.

L'intégration d'un message d'insertion d'une valeur v se déroule comme suit. L'identifiant de la valeur v_1 après laquelle v doit être insérée est récupéré à partir du message. Pour déterminer l'emplacement de v, le pair applique l'algorithme récursif suivant. S'il n'y a aucune valeur après v_1, alors v y est simplement placée. Sinon l'identifiant de v est comparé à l'identifiant de la valeur v_2 qui se situe juste après v_1. Si l'identifiant de v est supérieur à l'identifiant de v_2, alors v est placée entre v_1 et v_2. Dans le cas contraire, v doit être insérée après v_2. L'algorithme est donc répété en considérant v_2 comme v_1.

Pour illustrer cet algorithme, nous reprenons l'exemple de la figure 5.2a. Une fois l'ensemble des modifications intégrées, la séquence *RGA* de la figure 5.4 est obtenue. Un identifiant $\langle n, p_i \rangle$ est noté n_i. p_B débute avec la séquence '*lady*'. Il insère le caractère '*!*' entre '*y*' et la fin de la séquence. Il génère un identifiant dont l'horodatage est supérieur a ceux déjà générés dans la séquence. Il génère donc un horodatage égal à 5. L'intégration de cette insertion s'effectue simplement en plaçant le caractère après '*y*'. Il intègre ensuite l'insertion du caractère '*i*' qui référence le caractère '*d*'. Les caractères '*y*' et '*!*' se situent déjà après '*d*'. L'identifiant de '*i*' est supérieur à l'identifiant de '*y*' ($5_A > 4_A$). '*i*' est donc placé avant '*y*'. Il intègre de la même manière les insertions des caractères '*e*' et '*s*'.

La suppression d'une valeur et la lecture d'une séquence suivent les mêmes logiques que celles présentées pour *WOOT*. Une séquence *RGA* croît donc de manière monotone.

RGA utilise deux structures de données pour son implémentation : une liste chaînée et une table de hachage. La liste chaînée contient les valeurs et des pierres tombales ainsi

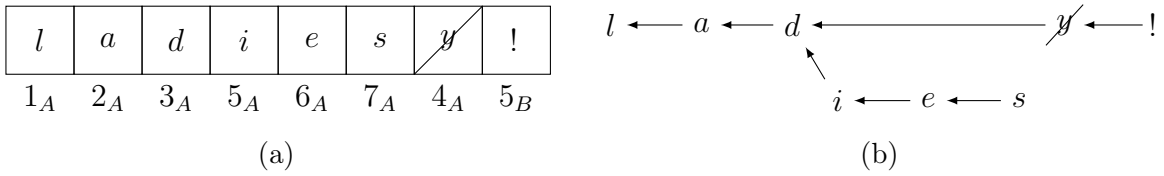


FIGURE 5.4 – Exemple d'une séquence *RGA*. (a) Séquence de valeurs et pierres tombales associées à leurs identifiants. Les pierres tombales sont représentées par un caractère barré. (b) Valeurs référencés par les valeurs de la séquence. Les références vers le début de la séquence sont omises.

que leurs identifiants. La table de hachage fait correspondre à un identifiant le chaînon de la liste chaînée qui inclut la valeur ou la pierre tombale qu'il identifie.

RGA repose sur une synchronisation par opérations. Les messages doivent être livrés exactement une fois. Les messages de suppression doivent être livrés après l'intégration du message d'insertion de la valeur à supprimer. Pour insérer une valeur, la valeur référencée doit déjà être présente dans la séquence. Un message d'insertion doit donc être intégré après l'intégration du message d'insertion de la valeur référencée. L'utilisation d'une seule référence permet de réduire le risque de bloquer une insertion dans l'attente que les valeurs qu'elle référence soient insérées.

RGA respecte la spécification forte [85]. Une extension [87] permet à *RGA* de rejeter tout entrelacement. Pour ce faire, un ensemble d'identifiants est associé à chaque valeur. Ils correspondent aux identifiants des valeurs qui référencent une même valeur au moment de l'insertion de la nouvelle valeur. Dans la figure 5.4, les valeurs, à l'exception de 'i' sont associées à un ensemble vide. Au moment de l'insertion du caractère 'i', le caractère 'y' référence déjà le caractère 'd'. 'i' inclut donc l'identifiant de 'y' dans l'ensemble qui lui est associé. L'intégration d'une insertion prend en compte cet ensemble pour déterminer l'ordre entre les valeurs. L'ajout de cet ensemble augmente l'occupation mémoire de la structure et a un coût en communication puisqu'il fait partie des messages d'insertion. En pratique, on s'attend à ce que la taille de ces ensembles soit faible [87].

Logoot [83] prend une approche distincte de *WOOT* et *RGA*. Une valeur ne référence pas d'autres valeurs. Une valeur peut donc être placée sans la présence des valeurs entre lesquelles elle a été insérée. Pour ce faire, chaque valeur est associée à une position unique et immuable. L'ordre entre les valeurs de la séquence est construit à partir de l'ordre total entre les positions. Lorsqu'une nouvelle valeur est insérée entre deux valeurs, sa position est générée de sorte à ce qu'elle soit comprise entre les positions de ces deux valeurs.

Une valeur peut toujours être insérée entre deux autres valeurs. L'ordre entre les positions est donc dense. *Logoot* génère un ordre total et dense à l'aide d'un ordre lexicographique entre des listes d'éléments. Par exemple l'ordre lexicographique des chaînes de caractères est un ordre total et dense. La chaîne ab est inférieure à la chaîne ac et la chaîne abx est comprise entre elles. Une nouvelle position est donc obtenue en recopiant éventuellement des éléments des positions voisines et en générant un nouvel élément. Dans le cas de la chaîne abx, les éléments a et b sont recopiés et x a été généré. Deux pairs peuvent insérer une valeur au même endroit. Pour assurer que des insertions parallèles

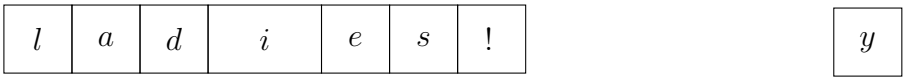


FIGURE 5.5 – Exemple d'une séquence *Logoot*. (a) Séquence de valeurs associées à leur position. (b) Valeur supprimée associée à sa position.

ne génèrent pas les mêmes positions, deux pairs ne doivent pas générer le même élément. Pour s'en assurer, *Logoot* utilise comme éléments des couples formés par un entier et l'identifiant du pair qui a généré le couple. Finalement pour éviter qu'un pair génère deux positions identiques, un nombre séquentiel est adjoint à la liste de couples. Chaque pair incrémente son propre nombre séquentiel après chaque exécution d'une opération d'insertion. Le nombre séquentiel est donc consommé par la génération d'une nouvelle position. Les positions *Logoot* présentées dans des travaux ultérieurs [23] sont légèrement différentes. La position devient une liste de triplets. Chaque triplet est constitué d'un entier aléatoire, de l'identifiant, et du nombre séquentiel du pair qui a généré le triplet. Nous nous basons sur la représentation originelle.

Deux positions sont comparées de manière lexicographique. Une position p_1 est supérieure à une position p_2 si et seulement si (i) leurs listes de couples sont identiques et le nombre séquentiel de p_1 est supérieur à celui de p_2, ou si (ii) la liste de couples de p_2 est un préfixe de la liste de couples de p_1, ou si (iii) les listes de p_1 et p_2 ont un préfixe commun de $n-1$ couples telles que le n-ième couple de p_1 est lexicographiquement supérieur au n-ième couple de p_2.

Une séquence *Logoot* est simplement une liste des valeurs auxquelles sont associées leur position. L'exécution d'une opération d'insertion d'une valeur v produit un message d'insertion composé de la valeur et de sa position. Pour générer sa position, les positions p_1 et p_2 des valeurs entre lesquelles v doit être insérée sont récupérées. La position de v doit être comprise entre ces deux positions. Le nombre séquentiel est déjà connu, il suffit de générer la liste de la nouvelle position. Pour simplifier l'explication, nous considérons que les positions p_1 et p_2 ont des listes de taille infinie. Ces listes de taille infinie sont obtenues en ajoutant des couples minimaux à p_1 et des couples maximaux à p_2. p_1 reste donc inférieur à p_2. La liste de la nouvelle position partage un préfixe commun de n couples ($n \geq 0$) avec la liste de p_1 tel qu'il ne soit pas possible de choisir un entier compris entre les entiers du k-ième couple de p_1 et du k-ième couple de p_2 avec $k \leq n$. Un $n+1$-ième couple est ajouté à la liste. Ce couple est généré par le pair qui insère v. Son entier est choisi aléatoirement entre les entiers du $n+1$-ième couple de p_1 et du $n+1$-ième couple de p_2. Le message est intégré à la copie et transmis aux autres pairs.

Pour illustrer cette algorithme, nous reprenons l'exemple de la figure 5.2a. Une fois l'ensemble des insertions intégrées, la séquence *Logoot* de la figure 5.5a est obtenue. Dans cette représentation un couple $\langle 2, p_A \rangle$ est représenté par b_A où b est la deuxième lettre de l'alphabet latin. Pour simplifier nous considérons donc que les entiers des couples sont compris entre 1 et 26. Par exemple, la position l_Aq_A5 a une liste de deux couples et un nombre séquentiel égal à 5. p_A insère le caractère '*i*' entre les caractères '*d*' et '*y*'. Il doit

donc générer une position entre les positions l_A3 et m_A4. l et m sont contiguës. Le couple l_A est donc copié. Le couple q_A est généré. Nous obtenons donc la liste l_Aq_A auquel est adjoint la valeur actuelle du nombre séquentiel de p_A. Puisqu'il s'agit de la cinquième insertion de p_A, son nombre séquentiel est égal à 5. Le caractère inséré est donc associé à la position l_Aq_{A5}. On constate un ordre total entre les positions des valeurs insérées : $f_A1 < h_A2 < l_A3 < l_Aq_A5 < m_A4 < p_A6 < r_A7 < t_B1$.

L'intégration d'un message d'insertion d'une valeur v de position p place v de manière à ce que les positions des valeurs qui précèdent v dans la séquence soient inférieures à p et les positions des valeurs qui succèdent v soient supérieures à p.

L'exécution d'une opération de suppression produit un message de suppression qui contient la position de la valeur à supprimer. L'intégration de ce message supprime définitivement la valeur associée à la position contenue dans le message.

Logoot repose sur une synchronisation par opérations. Les messages doivent être livrés exactement une fois. Un message qui supprime la valeur de position p est livré après le message qui insère la valeur de position p. Les messages d'insertion peuvent être livrés dans un ordre arbitraire.

Parce que les positions sont indépendantes les unes des autres, *Logoot* n'a pas besoin de pierres tombales. Toutefois la taille des positions croît au fur et à mesure des insertions de valeurs dans la séquence. Cette croissance de la taille des positions est particulièrement notable lorsque les insertions se concentrent dans une partie restreinte de la séquence.

Logoot respecte la spécification forte. Il souffre d'entrelacements [87].

Treedoc [22] repose sur la même idée que *Logoot* : placer les valeurs indépendamment des autres à l'aide de positions uniques et immuables. Cependant, *Treedoc* pose des contraintes sur les structures de données utilisées afin d'éviter le stockage des positions.

Les valeurs sont rangées au sein d'un arbre binaire. L'ordre de la séquence correspond à l'ordre infixe de l'arbre. La position d'une valeur correspond au chemin qui doit être emprunté dans l'arbre pour l'atteindre. Lorsqu'il n'y a pas d'insertions en parallèle, une position est donc simplement une liste de bits dans laquelle 0 indique de se diriger vers l'enfant gauche et 1 indique de se diriger vers l'enfant droit. L'ordre total et dense entre deux positions est formé par l'ordre lexicographique entre leurs listes de bits. Chaque insertion d'une nouvelle valeur crée un nœud situé à droite du nœud qui stocke la valeur qui la précède et à gauche du nœud de la valeur qui la succède.

L'insertion en parallèle de valeurs au même endroit conduit à la création de deux nœuds qui ont le même chemin dans l'arbre. Pour remédier à ce problème, *Treedoc* repose sur deux mécanismes : l'identification des valeurs et le regroupement des nœuds ajoutés en concurrence en super-nœud. Chaque valeur est identifiée par un *dot* constitué de l'identifiant du pair qui a inséré la valeur et du nombre séquentiel du pair. Un super-nœud est capable de stocker plusieurs valeurs. Les valeurs des super-nœuds sont ordonnées selon l'ordre lexicographique de leur *dot*. Chaque valeur d'un super-nœud est associée à un sous-arbre binaire. Lorsqu'une valeur v est insérée entre les valeurs v_1 et v_2 d'un super-nœud, v est insérée dans le sous-arbre associé à v_1. La position de v indique son emplacement dans le sous-arbre de v_1 en utilisant l'identifiant de v_1. Si la position de v_1 est 01 et que son identifiant est A_1, alors v a pour position $01A_1$.

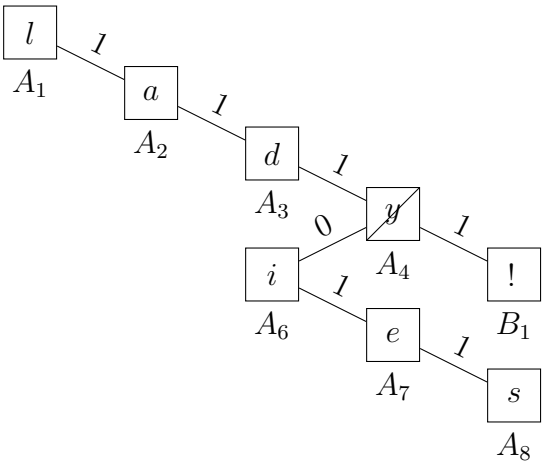


FIGURE 5.6 – Exemple d'une séquence *Treedoc* de valeurs associées à leur position (chemin dans l'arbre) et leur identifiant. Les nœuds qui sont des *pierres tombales* sont barrés.

L'exécution d'une opération d'insertion d'une valeur v entre deux valeurs v_1 et v_2 produit un message d'intégration qui inclut la valeur, l'identifiant de la valeur, et la position de la valeur dans l'arbre. La position est choisie de manière à ce qu'elle corresponde au chemin vers le nœud qui stockera v dans l'arbre. Ce nœud est à droite de v_1 et à gauche de v_2. L'intégration d'un message d'insertion s'effectue par la création d'un nœud dont le chemin correspond à la position de la valeur à insérer. Si des nœuds et des super-nœuds sont manquants sur le chemin, ils sont créés. Si le nœud où la valeur doit être stockée existe déjà, alors ce dernier est transformé en super-nœud.

L'exécution d'une opération de suppression de la n-ième valeur consiste à identifier l'emplacement de la valeur et à produire un message de suppression qui inclut l'identifiant de la valeur et sa position dans l'arbre. L'intégration d'un message de suppression consiste à déterminer la valeur à supprimer à l'aide de sa position et de son identifiant et à la supprimer. L'identifiant de la valeur permet de supprimer la bonne valeur lorsqu'elle se trouve dans un super-nœud. Le nœud qui stocke la valeur supprimée est conservé si et seulement s'il stocke une autre valeur (il s'agit d'un super-nœud) ou s'il a au moins un enfant. Si un super-nœud ne stocke plus qu'une seule valeur qui est associée à un arbre vide, alors le super-nœud est transformé en nœud simple.

La suppression de valeurs peut conduire à la présence de nœuds simples sans valeurs. Ces nœuds peuvent être apparentés à des *pierres tombales*. Similairement à *WOOT* et *RGA*, elles nuisent à l'efficacité des opérations exécutées et des messages intégrés sur la séquence. Toutefois, le plus grand inconvénient de *TreeDoc* réside dans le couplage de ses positions avec sa structure de données. Dans une écriture séquentielle de gauche à droite ou de droite à gauche, l'arbre devient une liste chaînée. La taille des positions croît rapidement.

La figure 5.6 correspond à la séquence *Treedoc* obtenue après l'exécution du scénario de la figure 5.2a. Le caractère 'y' a pour position 111 et identifiant A_4. Le caractère 'i' est inséré entre les caractères 'd' et 'y'. Un nœud doit donc être créé à gauche de 'y'. La position de 'i' est donc 1110.

Treedoc utilise une synchronisation par opérations. Les messages doivent être livrés exactement une fois et les messages de suppression d'une valeur doivent être livrés après l'insertion de cette valeur. Les messages d'insertion peuvent être livrés dans un ordre arbitraire.

LogootSplit [25] réduit l'occupation mémoire d'une séquence *Logoot* en groupant les valeurs par *blocs*. Pour ce faire, *LogootSplit* modifie la structure des positions de manière à ce qu'elles puissent être agrégées. Des valeurs contiguës dans la séquence dont les positions sont agrégeables sont groupées au sein d'un *bloc*.

Une position *LogootSplit* est une liste de quadruples. L'ordre total et dense des positions est formé par l'ordre lexicographique entre les listes de quadruples et les quadruples. Un quadruple est composé d'un entier choisi aléatoirement, de l'identifiant du pair qui a généré le quadruple, d'un nombre séquentiel, et d'un indice de bloc. Dans la figure 5.7, le caractère *'!'* est associé à une position qui contient le quadruple $\langle 20, B, 1, 0\rangle$. Ce quadruple est noté t_0^{B1} où t est la 20-ième lettre de l'alphabet latin. Deux positions sont agrégeables si et seulement si elles partagent la même liste de quadruples, à l'exception des derniers dont les indices de blocs sont différents mais contiguës. Par exemple, la position $f_2^{A1}q_0^{A2}$ est agrégeable à la position $f_2^{A1}q_1^{A2}$. Deux valeurs de la séquence peuvent être groupées en un bloc si et seulement si elles sont contiguës dans la séquence et leurs positions sont agrégeables. Dans la figure 5.7, les caractères *'l'*, *'a'*, et *'d'* sont contiguës et ont des positions agrégeables. Elles sont regroupées au sein d'un même bloc. Des positions agrégées sont représentées synthétiquement à l'aide de la position de la première valeur du bloc. La position d'une valeur peut alors être déduite de la première position du bloc et de son emplacement dans le bloc. Deux positions agrégeables sont nécessairement générées par le même pair. Un bloc est donc composé de valeurs insérées par un même pair. Ce pair est le propriétaire du bloc. Seul ce pair peut ajouter des valeurs au bloc. Dans la figure 5.7, p_A est le propriétaires des blocs *'lad'* et *'ies'*. p_B est le propriétaire du bloc *'!'*.

L'exécution d'une opération d'insertion d'une valeur v produit un message d'insertion qui contient v et la position de v. La position générée est comprise entre les positions p_1 et p_2 des valeurs v_1 et v_2 entre lesquelles v doit être insérée. L'algorithme de génération des positions favorise la génération de positions agrégeables. Il tente ainsi de produire une position agrégeable à p_1 ou p_2. S'il n'y parvient pas, il génère la position de la même manière que *Logoot*. Lorsque p_1 et p_2 sont agrégeables, p_1 devient un préfixe de la position de v.

Une séquence *LogootSplit* est une liste de blocs auxquels sont associés la position de la première valeur du bloc. L'intégration d'un message d'insertion d'une valeur v à une position p débute par l'identification de l'emplacement d'insertion. v doit être insérée entre les valeurs contiguës v_1 et v_2 de manière à ce que la position p_1 de v_1 soit inférieure à p et la position p_2 de v_2 soit supérieure à p. Nous différencions deux cas principaux : soit v_1 et v_2 font partie d'un même bloc, ou ils font partie de blocs distincts. Dans le premier cas, le bloc est scindé en deux blocs et la valeur est insérée entre ces deux blocs au sein d'un nouveau bloc. Dans le second cas, la valeur est soit insérée entre les deux blocs ou intégrée à l'un des blocs. Si p_1 et p sont agrégeables, alors v est ajoutée en tant que dernière valeur du bloc qui contient v_1. Sinon si p et p_2 sont agrégeables, alors v est ajoutée en tant que

FIGURE 5.7 – Exemple d'une séquence *LogootSplit*. (a) Séquence de valeurs associées à leur position. (b) Valeur supprimée associée à sa position.

première valeur du bloc qui contient v_2. Sinon, v est insérée entre les deux blocs au sein d'un nouveau bloc.

Pour illustrer la génération de positions et l'intégration de messages d'insertion, nous reprenons le scénario de la figure 5.2a. La séquence *LogootSplit* obtenue est représentée dans la figure 5.7a. Le pair p_A commence par une insertion contiguë de plusieurs caractères. Les positions sont générées de manière à ce qu'elles soient agrégeables. Nous obtenons ainsi le bloc qui contient la chaîne '*lady*'. p_A exécute ensuite l'opération a_2 qui insère le caractère '*i*' entre '*d*' et '*y*'. Aucune position agrégeable à la position de l'un des deux caractères ne peut être générée tel qu'elle soit comprise entre les positions des deux valeurs. La position générée a donc pour préfixe la position du caractère '*d*'. L'intégration de cette insertion produit la scission du premier bloc. En parallèle, le pair p_B insère le caractère '*!*' à la fin de la séquence après le bloc '*lady*'. Une position agrégeable à '*y*' ne peut être générée, étant donné que p_B n'est pas le propriétaire du bloc.

L'exécution d'une opération de suppression suit la même logique que *Logoot*. En revanche l'intégration d'un message de suppression peut conduire à la scission d'un bloc en deux. Pour éviter de générer plusieurs fois la même position, un bloc est associé à l'indice minimal et l'indice maximal des positions des valeurs qui ont été ajoutées au bloc.

L'insertion d'une valeur peut conduire à la scission d'un bloc. Bien que ce soit possible, *LogootSplit* ne propose pas de fusionner les deux blocs s'ils deviennent contigus. C'est une situation qui peut survenir lorsque les valeurs qui ont conduit à la scission du bloc sont supprimées.

LogootSplit utilise le même modèle de synchronisation que *Logoot*. Toutefois, pour simplifier son implémentation, il suppose que les insertions d'un même pair sont livrés dans le même ordre que leur génération.

Ce travail a été adapté pour *RGA*[27].

5.1.3 Catégorisation des séquences répliquées

Les CRDTs séquences attachent à chaque valeur des méta-données qui permettent de former l'ordre de la séquence. Ces méta-données sont générées de manière à garantir l'intention des insertions et des suppressions. Les séquences répliquées que nous avons présentées peuvent être réparties en deux approches :

L'approche à pierres tombales [21, 24] identifie chaque valeur avec un identifiant unique et immuable. Une valeur référence une des valeurs entre lesquelles elle est insérée ou éventuellement les deux. D'autres méta-données peuvent être ajoutées pour déterminer l'ordre total des valeurs de la séquence. Cette approche se caractérise généralement par une complexité spatiale constante des méta-données associées aux valeurs. Cependant, une valeur ne peut être insérée avant l'insertion des valeurs qu'elle référence. Cette approche est synchronisée par opérations et emploie une livraison causale des opérations. Elle requiert également la conservation des valeurs supprimées pour permettre l'insertion des valeurs qui les référencent. Pour ne pas apparaître lors des lectures, ces valeurs sont marquées et deviennent des *pierres tombales*. Parce qu'elles sont conservées au sein de la séquence répliquée, les pierres tombales réduisent les performances générales de la séquence répliquée lorsque le nombre de suppressions est important. *WOOT* et *RGA* utilisent cette approche.

L'approche à positions densément ordonnées [83, 22, 90] associe à chaque valeur une position unique et immuable. L'ensemble des positions est densément et totalement ordonné. Lorsqu'une valeur est insérée entre deux autres valeurs, sa position est générée de sorte à ce qu'elle soit comprise entre les positions de ces deux dernières. Les positions sont indépendantes : une valeur peut être insérée même si les valeurs entre lesquelles elle a été initialement insérée sont absentes. Une valeur et sa position peuvent donc être définitivement supprimées. Cependant, pour encoder un ordre dense, les positions n'ont pas une taille constante. Au fur et à mesure de la collaboration, la taille des positions croît. Ce qui réduit les performances de la séquence répliquée. *Logoot* et *Treedoc* utilisent cette approche.

Les deux familles de séquences répliquées réduisent toutes deux leur occupation mémoire en adoptant une approche à granularité variable [91, 25, 27]. Les valeurs sont groupées en blocs et leurs identifiants sont agrégés. La réduction est particulièrement notable lorsque les séquences sont utilisées pour l'édition collaborative [25, 27].

Plusieurs propositions tentent de réduire la taille des positions allouées aux valeurs pour l'approche à positions densément ordonnées. Nédelec et al. [90] proposent une stratégie de génération des positions qui assurent une complexité spatiale moyenne sous-linéaire. Cette stratégie a été évaluée sur des séquences à granularité fixe. Nicolas et al. [92] proposent de remplacer des positions par de nouvelles positions plus petites à l'aide d'un mécanisme de renommage.

La séquence répliquée que nous proposons dans la section suivante utilise l'approche à positions densément ordonnées.

5.2 Séquences répliquées synchronisées par différences

Nous proposons une approche générique qui permet d'obtenir des séquences répliquées synchronisées par différences d'états à partir de positions densément ordonnées. En plus de la croissance de la taille de leurs positions, les séquences obtenues peuvent souffrir d'une croissance linéaire des autres méta-données en fonction du nombre d'insertion. Nous proposons alors un nouveau type de positions qui permet de construire des séquences à granularité variable synchronisées par différences qui réduit la croissance de ces méta-données. Nous nommons ce protocole de séquences répliquées *Dotted LogootSplit*.

Dans un premier temps nous présentons un formalisme qui permet de décrire des positions densément ordonnées. Nous présentons alors un protocole générique synchronisé par opérations qui se base sur ce formalisme. Dans un second temps, nous utilisons ce formalisme pour introduire notre protocole générique synchronisé par différences d'états et *Dotted LogootSplit*.

5.2.1 Séquences répliquées à identifiants densément ordonnés

Dans cette sous-section, nous introduisons un formalisme qui nous permet de mieux appréhender la structure des positions densément ordonnées des séquences répliquées de la littérature. Il permet également de clarifier les contraintes de telles positions et d'ainsi ouvrir un espace de réflexion sur la structure des positions. Nous proposons également une implémentation générique pour les séquences répliquées synchronisées par opérations.

Les séquences répliquées à positions densément ordonnées indexent chaque valeur avec une unique position. L'ordre entre les valeurs de la séquence est induit par l'ordre total et strict entre leur position. Ces positions respectent plusieurs propriétés que nous décrivons dans les paragraphes suivants.

Dans une séquence, une valeur peut être insérée entre deux autres. De ce fait, il devrait toujours exister au moins une position entre tout couple de positions. De même, une valeur peut être insérée avant la première valeur de la séquence ou après la dernière valeur de la séquence. Pour toute position, il devrait donc exister au moins une position qui lui est inférieure et une position qui lui est supérieure. Pour simplifier, nous pouvons dire qu'il existe une position entre tout couple d'éléments de l'ensemble des positions étendu d'un élément minimal et d'un élément maximal. L'ensemble étendu des positions est donc équipé d'un ordre dense. Des exemples communs d'ordres denses sont l'ensemble des nombres réels et l'ensemble des nombres rationnels.

Pour illustrer notre propos, prenons l'exemple d'une séquence répliquée qui utilise des réels compris entre 0 et 1 en tant que positions. L'ensemble de positions étendu de l'élément minimal 0 et de l'élément maximal 1 forme un ensemble densément ordonné. Si un pair p_A insère une valeur dans une séquence vide, il choisit une position entre 0 et 1. Il peut par exemple choisir la position 0.1. Si p_A insère une valeur entre deux autres valeurs qui ont respectivement les positions 0.2 et 0.21, il peut par exemple choisir la position 0.201.

Dans un scénario distribué, les pairs peuvent insérer des valeurs de manière concurrente. Ils peuvent insérer ces valeurs entre le même couple de positions. Puisque les pairs ne se concertent pas, ils peuvent (accidentellement) choisir la même position. Pour éviter cette situation, l'ensemble des positions doit être partagé entre les pairs. Un pair choisit une position dans la partie qui lui appartient. L'ensemble doit être partagé de manière à respecter la propriété de densité de l'ordre. Tout pair est capable de choisir une position entre deux autres positions, même si ces positions ne sont pas incluses dans sa partie.

Nous reprenons l'exemple précédent. Supposons que la séquence est répliquée seulement par deux pairs : p_A et p_B. L'ensemble des positions peut être partagé de la manière suivante : p_A utilise uniquement des positions dont la partie décimale se termine par un chiffre impair et p_B utilise uniquement des positions dont la partie décimale se termine par un chiffre pair différent de zéro. Si p_A et p_B insèrent chacun une valeur en parallèle dans une séquence vide, ils ne peuvent pas choisir la même position. p_A peut par exemple choisir 0.1, alors que p_B choisit 0.2.

Lorsqu'une valeur est supprimée de la séquence, la position qui lui a été attachée pourrait être réutilisée pour l'insertion ultérieure d'une valeur. Cependant, la réutilisation de position peut conduire à des scénarios complexes tels que la suppression et l'insertion de deux valeurs distinctes avec la même position. Nous supposons donc qu'un pair ne réutilise pas de positions qu'il a préalablement choisies.

Supposons que p_A et p_B peuvent insérer au maximum quatre valeurs. Pour éviter la réutilisation d'une position préalablement utilisée, un pair peut utiliser une position dont la partie décimale se termine par un chiffre qui dépend du nombre d'insertions effectuées. Par exemple, si p_A insère sa n-ième valeur ($0 < n \leq 4$), il utilise le chiffre $n * 2 - 1$. Si p_B insère sa n-ième valeur, il utilise le chiffre $n * 2$.

Les protocoles de réplication utilisent généralement un identifiant localement unique au pair pour éviter de choisir deux fois la même position. L'ensemble des positions est donc également partagé par l'ensemble des identifiants locaux. De nouveau, ce partage ne doit pas compromettre la densité de l'ordre. Un pair peut choisir une position entre tout couple de positions avec l'identifiant local souhaité. La Definition 5.1 définit ce qu'est une position avec les contraintes que nous venons de discuter.

Dans la littérature, l'ensemble des entiers naturels est souvent utilisé comme identifiants locaux. En choisissant des nombres consécutifs pour chaque nouvelle position prise, il est possible de représenter de manière concise les identifiants locaux qui ne peuvent pas être utilisés pour la nouvelle position. En effet, au lieu de passer l'ensemble des nombres utilisés nous pouvons passer leur maximum. Nous parlons ainsi de nombres séquentiels. Lorsque les entiers naturels séquentiels sont utilisés comme identifiants locaux, conformément à la littérature, nous parlons de *dot* pour désigner l'identifiant $\mathsf{id}(p)$ d'une position p. Nous parlons alors de positions *dot*-ifiées. La Definition 5.2 définit ce qu'est une position *dot*-ifiée.

Definition 5.1 (Positions)**.** Soit Pos la famille des ensembles de positions. Un ensemble P est un ensemble de positions, et nous écrivons $P \in \mathsf{Pos}$, si et seulement si il est équipé des relations et des fonctions suivantes :

- $\mathsf{pid}(p) \in \mathbb{I}$ retourne l'identifiant globalement unique du pair qui génère la position p.
- $\mathsf{lid}(p) \in \mathsf{LID}_P$ retourne l'identifiant localement unique au pair $\mathsf{pid}(p)$ de la position p. LID_P correspond donc à l'ensemble des identifiants locaux. Il est dénombrable infini. Cet ensemble dépend de l'ensemble de positions P.
- $id(p) = \langle \mathsf{pid(p)}, \mathsf{lid(p)} \rangle$ retourne l'identifiant globalement unique de la position p.
- un ordre total strict et dense $<$ sur l'ensemble P étendu d'un élément minimal $\bot_P$ et d'un élément maximal $\top_P$. $\bot_P$ et $\top_P$ ne sont pas des éléments de P. $P_\bot$ et $P_\top$ dénotent respectivement l'ensemble P auquel est $\bot_P$ et l'ensemble P auquel est ajouté $\top_P$. $\bot_P$ est strictement inférieur à toutes les positions de P et à $\top_P$. $\top_P$ est strictement supérieur à toutes les positions de P et à $\bot_P$.

$$\forall l \in P_\bot \, \forall u \in P_\top \, \forall d \in \mathbb{I} \times \mathsf{LID}_P . \, l < u \implies \exists p \in P . \, l < p < u \wedge \mathsf{id}(p) = d$$

- $\mathsf{pick}_i(l, u, X)$ retourne une position qui appartient au pair d'identifiant i, qui n'utilise pas un identifiant local inclus dans l'ensemble fini X et qui est entre les positions l et u. l peut également prendre l'élément minimal $\bot_P$ de l'ordre précédemment défini. De même, u peut prendre l'élément maximal $\top_P$ de l'ordre précédemment défini.

$$\begin{aligned} \mathsf{pick}_{i \,\in\, \mathbb{I}} : \{l \in P_\bot\} \times \left\{u \in P^\top \;\middle|\; l < u\right\} \times \{X \in \mathcal{P}_{\mathsf{fin}}(\mathsf{LID_P})\} \\ \to \{p \in P \mid l < p < u \wedge \mathsf{id}(p) = \langle i, c \rangle \wedge c \notin X\} \end{aligned}$$

Definition 5.2 (Positions *dot*-ifiées)**.** Un ensemble P de positions est un ensemble de positions *dot*-ifiées, et nous écrivons $P \in \mathsf{DotPos}$, si et seulement si :

- L'ensemble des identifiants locaux correspond à l'ensemble des entiers naturels privé de 0.

$$\mathsf{LID}_P \stackrel{\text{def}}{=} \mathbb{N}^*$$

- La spécification de la fonction pick peut être raffinée comme suit :

$$\begin{aligned} \mathsf{pick}_{i \,\in\, \mathbb{I}} : \{l \in P_\bot\} \times \left\{u \in P^\top \;\middle|\; l < u\right\} \times \left\{X \in \mathcal{P}_{\mathsf{fin}}(\mathbb{N}^*)\right\} \\ \to \{p \in P \mid l < p < u \wedge \mathsf{id}(p) = \langle i, \max(X) + 1 \rangle\} \end{aligned}$$

Logoot [83] génère un ensemble de positions densément ordonnées à partir de listes d'éléments d'un ensemble ordonné et fini $\langle F, <_F \rangle$. F contient au moins deux éléments. L'ensemble de ces listes est équipé d'un ordre lexicographique et le dernier élément de chaque séquence est distinct de l'élément minimal de F. Nous dénotons par F^*, l'ensemble F privé de son élément minimal. Cet ensemble de listes est ainsi dénoté par $\bigcup\{F^n \times F^* \mid n \in \mathbb{N}\}$, que nous abrégeons en $F^{n \in \mathbb{N}} \times F^*$. L'ordre lexicographique est défini comme suit :

$$\begin{aligned} &\forall n, m \in \mathbb{N}_0 \; \forall a \in F^n \times F^* \; \forall b \in F^m \times F^*. \\ &\quad a \leq_{\mathsf{lex}} b \iff \exists k \leq m . \, (\forall j < k . \, a_j = b_j) \wedge (k = n + 1 \vee a_k <_F b_k) \end{aligned} \tag{5.1}$$

Si nous prenons $F \stackrel{\text{def}}{=} \{0, 1\}$ avec $0 < 1$, alors la liste $\langle 0, 1\rangle$ précéde la liste $\langle 1, 1\rangle$. Ces deux positions liste $\langle 1, 1, 1\rangle$. Le dernier élément de chaque liste est bien différent de l'élément minimal de F de sorte à ce qu'il existe toujours une liste plus petite que la liste considérée.

Logoot utilise un ensemble F plus complexe pour respecter les contraintes des positions que nous avons précédemment discutées. Une position *Logoot* est une paire qui consiste d'une liste de couples et d'un nombre séquentiel. Le premier élément de chaque couple est un nombre prioritaire et le second élément est l'identifiant du pair qui est utilisé comme désambiguïteur quand deux pairs choisissent en concurrence le même nombre prioritaire. Le nombre séquentiel est incrémenté après chaque insertion locale et évite ainsi la réutilisation de positions attachées à des valeurs supprimées. Une position *Logoot* est donc *dot*-ifiée. Une position est identifiée de manière unique avec le *dot* qui est construit à partir de l'identifiant du pair inclus dans le dernier couple de la position et son nombre séquentiel. Deux positions sont ordonnées à partir de l'ordre lexicographique entre leur composante. L'ordre est dense, étant donné l'extensibilité de la liste et l'exclusion du nombre prioritaire minimal dans le dernier couple de la séquence. Les nombres prioritaires sont des entiers naturels strictement inférieur à u. u est supérieur ou égal à 2. Dans l'équation 5.2, $((\mathbb{N}_0^{<u} \times \mathbb{I})^{n\in\mathbb{N}_0} \times (\mathbb{N}^{*<u} \times \mathbb{I}))$ correspond à l'ensemble des listes de couples. Le dernier couple exclut bien le nombre prioritaire minimal.

$$\mathsf{LogootPos}_{u\,\in\,\mathbb{N}^{>1}} \stackrel{\text{def}}{=} ((\mathbb{N}_0^{<u} \times \mathbb{I})^{n\in\mathbb{N}_0} \times (\mathbb{N}^{*<u} \times \mathbb{I})) \times \mathbb{N}^* \tag{5.2}$$

$$\mathsf{id}(\langle\{\langle \_, i_k\rangle\}_{k=0}^{m}, \mathsf{n}\rangle) \stackrel{\text{def}}{=} \langle i_m, n\rangle \tag{5.3}$$

Dans l'état de l'art présenté, l'ensemble des nombres prioritaires correspond à l'alphabet latin. Si un pair p_A insère sa deuxième valeur entre les positions $\langle\langle f_A\rangle, 1\rangle$ et $\langle\langle g_B\rangle, 1\rangle$, il peut choisir la position $\langle\langle f_A, h_A\rangle, 2\rangle$. f_A correspond au premier couple de la liste de la position. f est le nombre prioritaire et A identifie le paire p_A. La liste de couples est suivie du nombre séquentiel 2.

Treedoc [22] emploie une autre structure de positions. Il génère un ensemble densément ordonné à l'aide d'un n-uplet de *bits* et de *dots*. Une position *Treedoc* est une paire. Le premier élément de la paire est un n-uplet de *bits* et de *dots*. Le n-uplet se termine toujours par un *bit*. Le second élément de la paire est un *dot* qui identifie de manière unique la valeur. Une position *Treedoc* est *dot*-ifiée. L'ordre entre deux positions correspond à l'ordre lexicographique de ces composantes. Lorsque deux valeurs sont insérées en parallèle au même endroit, elles ont le même n-uplet de *bits* et de *dots*. Par exemple les valeurs associées aux positions $\langle\langle 0, 1, 0\rangle, A_1\rangle$ et $\langle\langle 0, 1, 0\rangle, B_1\rangle$ ont été insérées en parallèle au même emplacement. Leur *dot* respectif sert alors de désambiguïteur. Si une valeur est insérée entre ces deux valeurs, le *dot* de la valeur à gauche fera partie du n-uplets de la position nouvellement générée. Par exemple la valeur associée à la position $\langle\langle 0, 1, 0, A_1\rangle, A_2\rangle$ se situe entre les deux valeurs précédentes.

$$\mathsf{TreedocPos} \stackrel{\text{def}}{=} ((\mathbb{I} \times \mathbb{N}^*)^{m\in\{0,1\}} \times \{0, 1\})^{n\in\mathbb{N}_0} \times (\mathbb{I} \times \mathbb{N}^*) \tag{5.4}$$

$$\mathsf{id}(\langle \_, \langle i, n\rangle\rangle) \stackrel{\text{def}}{=} \langle i, n\rangle \tag{5.5}$$

Dans le chapitre 3, nous avons vu que les types de données répliquées pouvaient être synchronisés avec différentes approches. Le choix du mécanisme de synchronisation précise l'implémentation du type de données.

La figure 5.8 propose une implémentation générique d'une séquence à positions densément ordonnées synchronisée par opérations. L'équation 5.7 précise l'ensemble des états du CRDT. Un état est défini comme un couple $\langle m, x\rangle$ où x est un ensemble fini d'identifiants locaux, et m est un ensemble fini d'associations de positions et de valeurs. L'association d'une position p et d'une valeur v est représentée par un couple $\langle p, v\rangle$. x contient l'ensemble des identifiants locaux utilisés par le pair qui détient l'état de la séquence. Si des positions *dot*-ifiées sont utilisées, x peut être remplacé par un nombre séquentiel.

L'exécution d'une insertion ou d'une suppression se déroule en deux étapes. Un message est d'abord généré à l'aide de la fonction $\mathsf{prepare}$. L'appel $\mathsf{prepare}_i(\sigma, o)$ retourne un message généré par l'exécution de l'opération de modification o par le pair p_i sur l'état σ. Le message est ensuite intégré à l'aide de la fonction $\mathsf{integrate}$. L'appel $\mathsf{integrate}_i(\sigma, \mathsf{msg})$ retourne la mise-à-jour de l'état σ qui correspond à σ auquel a été intégré le message msg sur le pair p_i. Le message est également transmis aux autres pairs qui l'intégreront à terme.

Pour faciliter l'implémentation de ces fonctions, nous définissions les fonctions $\mathsf{posIndex}$ et nthPos. $\mathsf{posIndex}(m, p)$ retourne l'indice d'une position p dans la séquence. Pour déterminer cette indice, il suffit de compter le nombre de positions de la séquence qui sont inférieures à p. $\mathsf{nthPos}(m, k)$ retourne la position de la k-ième valeur de l'ensemble m des associations des valeurs et de leur position. $|m|$ correspond au cardinal de l'ensemble des associations m. Il s'agit donc de la taille de la séquence.

$$
\begin{aligned}
&\forall V\ \forall P \in \mathsf{Pos}\ \forall m : P \hookrightarrow V.\, \mathsf{dom}(m)\ \textit{is finite} \implies \\
&\quad \mathsf{posIndex}(m, p) \stackrel{\text{def}}{=} |\{l \in \mathsf{dom}(m) \mid l <_P p\}| \\
&\quad \mathsf{nthPos}(m, k) \stackrel{\text{def}}{=} p \quad \textbf{if } k < |m| \\
&\qquad \textbf{where } p \in \mathsf{dom}(m) \wedge \mathsf{posIndex}(m, p) = k
\end{aligned}
\tag{5.6}
$$

L'évaluation d'une opération de lecture retourne la liste des valeurs de la séquence (équation 5.8). L'ordre des valeurs dans la liste correspond à l'ordre de leur position dans la séquence. Le message d'insertion généré par l'exécution d'une opération d'insertion d'une valeur v avant la n-ième valeur de la séquence correspond à un triplet constitué de l'étiquette ins, de la position choisie pour v, et de la valeur v (équation 5.9). Pour générer la position de v, nous déterminons les positions entre lesquelles v est insérée. Si la séquence est vide, il s'agit des positions $\bot_P$ et $\top_P$. Si v est insérée avant la première valeur de la séquence, alors elle est insérée entre la position $\bot_P$ et la position de la première valeur. Si v est insérée après la dernière valeur de la séquence, alors elle est insérée entre la position de la dernière valeur et la position $\top_P$. Finalement si v est insérée entre deux valeurs de la séquence, alors elle est insérée entre les positions de ces deux valeurs. L'intégration d'un message d'insertion d'une valeur v à une position p ajoute l'association $\langle p, v\rangle$ à l'ensemble des associations m de la séquence (équation 5.11). Si le message a été généré par le pair qui intègre le message, alors l'identifiant local de p est enregistré dans x pour éviter sa réutilisation ultérieure. Le message de suppression généré par l'exécution d'une opération de suppression de la n-ième valeur correspond à un couple constitué de l'étiquette del et

de la position de la valeur à supprimer (équation 5.10). Si la séquence est vide, le message retournée est vide ($\bot_{\mathsf{Msg}}$). L'intégration d'un message de suppression d'une valeur à une position p supprime l'association de m qui contient la position p (équation 5.12). Cette association doit exister.

$$\forall V\ \forall P \in \mathsf{Pos}.\,\mathsf{OpPosSeq}\langle P, V\rangle \stackrel{\text{def}}{=} \{m : P \hookrightarrow V\} \times \mathcal{P}_{\mathsf{fin}}(\mathsf{LID}_P) \quad \textbf{where } \mathsf{dom}(m) \textit{ is finite} \tag{5.7}$$

$$\mathsf{eval}_i(\langle m, x\rangle, \mathsf{rd}) \stackrel{\text{def}}{=} \{v_k\}_{k=0}^{k=|m|-1} \quad \textbf{where } \langle p_k, v_k\rangle \in m \quad \textbf{where } p_k \stackrel{\text{def}}{=} \mathsf{nthPos}(m, k) \tag{5.8}$$

$$\mathsf{prepare}_i(\langle m, x\rangle, \mathsf{ins}(j, v)) \stackrel{\text{def}}{=} \langle \mathbf{ins}, p, v\rangle \quad \textbf{where } p \stackrel{\text{def}}{=} \mathsf{pick}_i(l, u, x) \quad \textbf{where } l \stackrel{\text{def}}{=} \begin{cases} \bot_P & \textbf{if } k = 0 \\ \mathsf{nthPos}(m, k-1) \end{cases} \quad \textbf{where } u \stackrel{\text{def}}{=} \begin{cases} \top_P & \textbf{if } k = |m| \\ \mathsf{nthPos}(m, k) \end{cases} \quad \textbf{where } k \stackrel{\text{def}}{=} \min(j, |m|) \tag{5.9}$$

$$\mathsf{prepare}_i(\langle m, x\rangle, \mathsf{del}(j)) \stackrel{\text{def}}{=} \begin{cases} \langle \mathbf{del}, p\rangle & \textbf{if } |m| \neq 0 \\ \bot_{\mathsf{Msg}} \end{cases} \quad \textbf{where } p \stackrel{\text{def}}{=} \mathsf{nthPos}(m, \min(j, |m|)) \tag{5.10}$$

$$\mathsf{integrate}_i(\langle m, x\rangle, \langle \mathbf{ins}, p, v\rangle) \stackrel{\text{def}}{=} \begin{cases} \langle m \cup \{\langle p, v\rangle\}, x \cup \{c'\}\rangle & \textbf{if } i = i' \\ \langle m \cup \{\langle p, v\rangle\}, x\rangle \end{cases} \quad \textbf{where } \langle i', c'\rangle \stackrel{\text{def}}{=} \mathsf{id}(p) \tag{5.11}$$

$$\mathsf{integrate}_i(\langle m, x\rangle, \langle \mathbf{del}, p\rangle) \stackrel{\text{def}}{=} \langle m - \{\langle p, m(p)\rangle\}, x\rangle \tag{5.12}$$

FIGURE 5.8 – Implémentation générique d'une séquence à positions densément ordonnées synchronisée par opérations

Les séquences répliquées que nous venons de présenter sont à granularité fixe. Ils ont un surcoût important car chaque valeur est associée à une méta-donnée qui correspond à sa position. La littérature propose différentes approches pour réduire ce surcoût [25, 27, 90, 92]. Dans la sous-section suivante nous introduisons l'approche à granularité variable.

5.2.2 Séquences répliquées à granularité variable

Les séquences répliquées précédemment présentées associent à chaque valeur une position. Dans le cas d'édition collaborative de texte, les valeurs sont des caractères. Le caractère prend moins de mémoire que la position qui lui est associée. Le support de l'édition collaborative induit donc un coût important en mémoire. Ce coût se retrouve également en communication puisque chaque message d'insertion contient la valeur et sa position.

Au lieu d'utiliser des caractères en tant que valeurs, certains travaux [21] soulignent que l'utilisation de chaînes de caractères en tant que valeurs est possible. Cette approche à le désavantage de ne pas permettre l'insertion de caractères au sein de ces chaînes de caractères. Ce n'est donc pas une approche généralisable à toute applications d'édition collaborative de texte.

L'approche à granularité variable propose de permettre l'agrégation de valeurs et de leur position en blocs. Elles peuvent donc être représentées efficacement en mémoire tout en autorisant leur séparation en plusieurs blocs. Chaque bloc est indexé par la position associée à la première valeur. La position de chaque valeur est inférée à partir de cette dernière et de leur indice dans le bloc.

Prenant une position p, il existe une unique position q qui est strictement supérieure et agrégeable à q. Par simplicité nous supposons que p est l'unique position qui est inférieure à q et agrégeable à q. Par conséquent toute position entre p et q ne peuvent être agrégé ni avec p ni avec q. Il s'en suit que l'insertion d'une valeur entre deux valeurs qui appartiennent à un bloc conduit à la division du bloc en deux blocs. Ce type d'insertion produit ainsi trois blocs.

Definition 5.3 (Positions agrégeable)**.** Un ensemble P de positions est un ensemble de positions *agrégeables*, et nous écrivons $P \in \mathsf{AggPos}$, si et seulement si l'ensemble est équipé d'une fonction bijective agg qui retourne l'unique position agrégeable et strictement supérieure à la position passée en paramètre.

$$\mathsf{agg} : \{p \in P\} \to \{q \in P \mid p <_P q\}$$

Théorème 5.1 (Réduction d'une chaîne de positions agrégeables)**.** *Une chaîne de positions agrégeables peut être déduite à partir de sa première position et de sa longueur.*

Démonstration. Nous procédons par induction. La première position est connue. Supposons que nous connaissions la n-ième position de la chaîne que nous notons p. $\mathsf{agg}(p)$ est l'unique position agrégeable à p et strictement supérieure à p. Elle est la $n+1$-ième position de la chaîne. A partir de la première position q et de la taille m de la chaîne nous pouvons ainsi obtenir la chaîne $\{p_k \in P \mid 0 \leq k < m \wedge p_0 = q \wedge \forall 0 \leq i < m-1.\, \mathsf{agg}(p_i) = p_{i+1}\}$. □

Yu [91] propose le premier type de séquences répliquées à granularité variable. Cependant, *LogootSplit* [25] est le premier type de séquences répliquées qui exploite l'idée d'agréger des positions. Une position *LogootSplit* est une liste de n-uplets. Le premier élément d'un n-uplet est un nombre prioritaire, le second élément est l'identifiant du pair qui a généré le n-uplet, le troisième élément est un nombre séquentiel, et le dernier élément est l'indice de la valeur dans le bloc. *LogootSplit* est construit sur les idées introduites par

Logoot. Il utilise le nombre prioritaire pour ordonner les positions. Dans le cas où deux positions partagent le même nombre prioritaire, elles sont ordonnées par l'identifiant du pair. Les identifiants des pairs agissent donc comme des désambiguïteurs. En revanche, il utilise des *dots* pour identifier de manière unique un bloc au lieu d'une valeur. A chaque fois qu'un nouveau bloc est généré, le nombre séquentiel est incrémenté. Par exemple, le premier bloc du pair p_A est identifié par le *dot* A_1, et le second par A_2. Une position et par extension la valeur qui lui est associée sont identifiées de manière unique par le *dot* du bloc correspondant et l'indice de la valeur dans le bloc. Par exemple, $\langle A_1, 0\rangle$ et $\langle A_1, 1\rangle$ identifient les deux premières valeurs du premier bloc du pair p_A. *LogootSplit* permet d'ajouter des valeurs en début et en fin de blocs. Il génère ainsi les positions de sorte à ce qu'elles soient respectivement agrégeable avec la première et la dernière position. Pour ce faire, *LogootSplit* utilise des indices négatifs et positifs. Par exemple si le pair p_A insère une valeur en début de son premier bloc, la valeur est identifiée par $\langle A_1, -1\rangle$. Quand une valeur est insérée entre deux valeurs d'un même bloc, le bloc est divisé. La position générée inclut l'ensemble des n-uplets de la position qui la précède, auquel est joint un nouveau n-uplet.

$$\mathsf{LogootSplitPos}_u \stackrel{\text{def}}{=} (\mathbb{N}_0^{<u} \times \mathbb{I} \times \mathbb{N}_0 \times \mathbb{Z})^{n\in\mathbb{N}^*} \times \mathbb{N}^{*<u} \times \mathbb{I} \times \mathbb{N}^* \times \mathbb{Z} \tag{5.13}$$

$$\mathsf{id}(\{\langle \_, i_k, n_k, z_k\rangle\}_{k=0}^{m}) \stackrel{\text{def}}{=} \langle i_m, \langle n_m, z_m\rangle\rangle \tag{5.14}$$

$$\mathsf{agg}(\{\langle x_k, i_k, n_k, z_k\rangle\}_{k=0}^{m}) \stackrel{\text{def}}{=} \{a_h\}_{h=0}^{m}$$
$$\textbf{where } a_h = \begin{cases} \langle x_m, i_m, n_m, z_m + 1\rangle & \textbf{if } h = m \\ \langle x_h, i_h, n_h, z_h\rangle \end{cases} \tag{5.15}$$

Nous disposons des concepts nécessaires à la présentation de notre séquence répliquée.

5.2.3 Approche générique

Dans cette sous-section, nous proposons une approche générique qui permet de synchroniser par différences d'états une séquence répliquée à positions densément ordonnées qui respecte le formalisme présenté dans les sous-sections précédentes.

Les types de données répliqués synchronisés par différences d'états représentent les modifications sous la forme de différences d'états. L'intégration d'une différence d'états consiste donc à la fusion de deux états. Les états peuvent être fusionnés dans un ordre quelconque et plusieurs fois. Cette synchronisation est donc parfaitement adaptée à des systèmes de passage de messages non-fiables où les messages peuvent être dupliqués et réordonnées.

Les séquences répliquées à positions densément ordonnées de la littérature sont synchronisées par opérations. Les opérations doivent être intégrées dans un ordre spécifique et exactement une fois. Cependant, seuls les couples d'opérations qui concernent une même valeur ont un ordre d'intégration spécifique : l'insertion d'une valeur doit être intégrée avant sa suppression.

Si nous considérons une séquence qui autorise uniquement des insertions, elle peut donc être aisément synchronisée par différences d'états. Pour ce faire, il suffit d'insérer

une valeur et sa position si et seulement si elle n'est pas déjà présente dans la séquence. Nous obtenons ainsi la propriété d'idempotence des séquence synchronisées par différences d'états : une insertion peut être intégrée plusieurs fois.

Lorsque la séquence autorise les suppressions, des valeurs peuvent être supprimées. Si une insertion d'une valeur est intégrée après sa suppression, la valeur est alors de nouveau insérée. L'intention de la suppression est violée. Pour s'assurer qu'une valeur supprimée est définitivement supprimée, et ce même lorsque son insertion est intégrée ultérieurement, nous devons garder une trace de l'ensemble des valeurs qui ont été insérées dans la séquence. Nous employons un *contexte causal* [18] qui inclut l'identifiant unique de chaque valeur insérée dans la séquence. Nous définissons donc le contexte causal comme un ensemble fini d'identifiants de valeurs (équation 5.16). Un identifiant d'une valeur correspond à un couple $\langle i, l \rangle$ où i est l'identifiant du pair qui a généré l'identifiant, et l est un identifiant local à ce pair. La fonction lids donne l'ensemble des identifiants locaux présents dans le contexte causal pour un pair donné (équation 5.17).

$$\mathsf{CausalContext}\langle P \rangle \stackrel{\text{def}}{=} \mathcal{P}_{\mathsf{fin}}(\mathbb{I} \times \mathsf{LID}_P) \tag{5.16}$$

$$\mathsf{lids}_i(c) \stackrel{\text{def}}{=} \{n \mid \langle i, n \rangle \in c\} \tag{5.17}$$

Lorsqu'une insertion est intégrée, nous vérifions la présence ou l'absence de l'identifiant de la valeur dans le contexte causal. Si l'identifiant est présent, la valeur a déjà été insérée et éventuellement été supprimée. Dans ce cas, la séquence reste inchangée. Si l'identifiant est absent, la valeur n'a pas encore été insérée. Elle est donc insérée et son identifiant est ajouté au contexte causal.

Lorsqu'une suppression est intégrée, nous différencions deux cas. Si l'identifiant de la valeur à supprimer est présent dans le contexte causal, alors nous nous assurons que la valeur n'est plus présente dans la séquence. Si elle l'est nous la supprimons. Si l'identifiant de la valeur à supprimer est absent du contexte causal, alors nous l'ajoutons simplement au contexte causal.

La figure 5.9 présente l'implémentation générique d'une séquence synchronisée par différences d'états. Nous donnons un exemple de l'exécution de cette implémentation dans la sous-section suivante. Nous représentons un état d'une telle séquence par un couple $\langle m, c \rangle$ où m associe les positions aux valeurs de la séquence et c est le contexte causal de la séquence (équation 5.18). Les messages de synchronisation sont des différences d'états. L'exécution de l'insertion d'une valeur v à l'indice j génère une position p comprise entre les positions associées aux $j-1$-ième et j-ième valeurs de la séquence. Le message d'insertion consiste donc en un couple qui contient uniquement l'association de p à la valeur v et un contexte causal qui contient uniquement l'identifiant unique dérivé de la position (équation 5.20). L'exécution de la suppression de la j-ième valeur génère un message de suppression qui contient aucune association et un contexte causal qui contient uniquement l'identifiant de la valeur à supprimer (équation 5.21).

La fusion d'un état $\sigma_1 \stackrel{\text{def}}{=} \langle m, c \rangle$ avec un état $\sigma_2 \stackrel{\text{def}}{=} \langle m', c' \rangle$ produit un état $\sigma_3 \stackrel{\text{def}}{=} \langle m'', c'' \rangle$ (équation 5.22). Le contexte causal de σ_3 correspond à l'union des contextes causaux de σ_1 et σ_2. Les associations de σ_3 correspondent à l'intersection de l'ensemble des associations de σ_1 et de l'ensemble des associations de σ_2, ainsi que des associations de σ_1 et σ_2

dont l'identifiant n'est pas présent respectivement dans le contexte causal de σ_2 et le contexte causal de σ_1. Ces deux derniers ajouts permettent d'insérer les valeurs présentes uniquement dans l'un des deux états tout en s'assurant de ne pas insérer des valeurs supprimées dans l'un des deux états.

$$\begin{aligned}
&\forall V\ \forall P \in \mathsf{Pos.\,DeltaSeq}\langle P, V\rangle \stackrel{\text{def}}{=} \{m : P \hookrightarrow V\} \times \mathcal{P}_{\mathsf{fin}}(\mathbb{I} \times \mathsf{LID}_P) \\
&\qquad \textbf{where } \mathsf{dom}(m) \textit{ is finite}
\end{aligned} \tag{5.18}$$

$$\begin{aligned}
&\mathsf{eval}_i(\langle m, c\rangle, \mathsf{rd}) \stackrel{\text{def}}{=} \{v_k\}_{k=0}^{k=|m|-1} \\
&\qquad \textbf{where } \langle p_k, v_k\rangle \in m \\
&\qquad \textbf{where } p_k \stackrel{\text{def}}{=} \mathsf{nthPos}(m, k)
\end{aligned} \tag{5.19}$$

$$\begin{aligned}
&\mathsf{prepare}_i(\langle m, c\rangle, \mathsf{ins}(j, v)) \stackrel{\text{def}}{=} \langle\{\langle p, v\rangle\}, \{\mathsf{id}(p)\}\rangle \\
&\qquad \textbf{where } p \stackrel{\text{def}}{=} \mathsf{pick}_i(l, u, \mathsf{lids}_i(c)) \\
&\qquad \textbf{where } l \stackrel{\text{def}}{=} \begin{cases} \bot_P & \textbf{if } k = 0 \\ \mathsf{nthPos}(m, k-1) & \end{cases} \\
&\qquad \textbf{where } u \stackrel{\text{def}}{=} \begin{cases} \top_P & \textbf{if } k = |m| \\ \mathsf{nthPos}(m, k) & \end{cases} \\
&\qquad \textbf{where } k \stackrel{\text{def}}{=} \min(j, |m|)
\end{aligned} \tag{5.20}$$

$$\begin{aligned}
&\mathsf{prepare}_i(\langle m, c\rangle, \mathsf{del}(j)) \stackrel{\text{def}}{=} \begin{cases} \langle\emptyset, \{\mathsf{id}(p)\}\rangle & \textbf{if } |m| \neq 0 \\ \langle\emptyset, \emptyset\rangle & \end{cases} \\
&\qquad \textbf{where } p \stackrel{\text{def}}{=} \mathsf{nthPos}(m, \min(j, |m|))
\end{aligned} \tag{5.21}$$

$$\begin{aligned}
&\mathsf{integrate}_i(\langle m, c\rangle, \langle m', c'\rangle) \stackrel{\text{def}}{=} \langle m'', c \cup c'\rangle \\
&\qquad \textbf{where } m'' \stackrel{\text{def}}{=} (m \cap m') \\
&\qquad\quad \cup \{\langle p, v\rangle \in m \mid \mathsf{id}(p) \notin c'\} \cup \{\langle p', v'\rangle \in m' \mid \mathsf{id}(p') \notin c\}
\end{aligned} \tag{5.22}$$

FIGURE 5.9 – Implémentation générique d'une séquence synchronisée par différences

Les protocoles de synchronisation par différences d'états assurent généralement le modèle de cohérence causal. Le protocole de réplication peut utiliser cette garantie pour représenter plus efficacement le contexte causal pour certains types de positions. Une position *Logoot* utilise un nombre séquentiel comme identifiant local. De ce fait, le contexte causal peut être efficacement représenté par un vecteur de versions [68, 69]. Ce vecteur associe à chaque pair l'identifiant de la dernière valeur insérée par le pair correspondant. D'autres positions, comme les positions *LogootSplit* ne permettent pas cette optimisation. Le contexte causal croît linéairement au fur et à mesure de l'insertion de valeurs. Cependant, contrairement aux positions *Logoot*, les positions *LogootSplit* peuvent être agrégées. Dans la sous-section suivante, nous présentons un nouveau type de position qui permet à la fois une représentation efficace du contexte causal et l'agrégation de positions.

5.2.4 Dotted LogootSplit

Les positions *Logoot* permettent de représenter efficacement le contexte causal par un vecteur de versions. Pour ce faire, chaque valeur est identifiée par un *dot* : un couple qui consiste d'un pair et d'un nombre séquentiel. Les positions *LogootSplit* ne permettent pas cette optimisation. Chaque valeur est identifiée par un couple qui contient un *dot* et un indice. Le *dot* identifie de manière unique le bloc dans lequel la valeur se trouve. L'indice correspond alors à l'indice de la valeur dans ce bloc.

Nous proposons un nouveau type de positions qui repose sur les idées introduites par *Logoot* et *LogootSplit*. Similairement à ces deux approches, nous utilisons des listes de n-uplets ordonnées par leur ordre lexicographique. Chaque n-uplet consiste d'un nombre prioritaire, de l'identifiant du pair qui a généré le n-uplet, et d'un nombre séquentiel. Similairement à *Logoot*, chaque valeur est identifiée par un *dot*. Chaque fois qu'un pair exécute une insertion, il incrémente son nombre séquentiel et assigne un nouveau *dot* à la valeur insérée. Ce *dot* est constitué de l'identifiant du pair et du nombre séquentiel du dernier n-uplet de la liste. Ces positions sont donc *dot*-ifiées. Par exemple, la position $\langle\langle f, A, 3\rangle, \langle q, A, 5\rangle\rangle$ est composée de deux n-uplets : $\langle f, A, 3\rangle$ et $\langle q, A, 5\rangle$. Les nombres prioritaires correspondent ici à des lettres latines. Le dernier n-uplet indique le *dot* de la position : $\langle A, 5\rangle$. Il s'agit donc de la cinquième position générée par le pair p_A. Pour simplifier les exemples, cette position est abrégée par $f_3^A q_5^A$.

Deux positions sont agrégeables, si et seulement si (i) elles divergent uniquement par leur identifiant local (le nombre séquentiel du dernier n-uplet), et (ii) elles ont des identifiants locaux contiguës. Le nombre séquentiel joue à la fois le rôle d'identifiant local et d'indice de bloc. Par exemple, f_3^A est agrégeable à f_4^A et $f_3^A q_5^A$ est agrégeable à $f_3^A q_6^A$.

Pour assurer l'ordre dense des positions, le nombre prioritaire du dernier n-uplet d'une position est différent du nombre prioritaire minimal. Les équations suivantes définissent l'ensemble des positions *Dotted LogootSplit*.

$$\mathsf{DottedLogootSplitPos}_u \stackrel{\text{def}}{=} \left(\mathbb{N}_0^{<u} \times \mathbb{I} \times \mathbb{N}^*\right)^{n\in\mathbb{N}_0} \times \left(\mathbb{N}^{*<u} \times \mathbb{I} \times \mathbb{N}^*\right) \tag{5.23}$$

$$\mathsf{agg}(\{\langle x_k, i_k, n_k\rangle\}_{k=0}^m) \stackrel{\text{def}}{=} \{a_h\}_{h=0}^m$$
$$\textbf{where } a_h = \begin{cases} \langle x_m, i_m, n_m + 1\rangle & \textbf{if } h = m \\ \langle x_h, i_h, n_h\rangle \end{cases} \tag{5.24}$$

L'utilisation de l'implémentation de la figure 5.9 et de ce nouveau type de positions permettent la réplication d'une séquence *Dotted LogootSplit* synchronisée par différences d'états. Afin d'illustrer ce nouveau CRDT, nous reprenons le scénario de la figure 5.2a. La figure 5.10 décrit les états successifs de la copie du pair p_A. La figure 5.11 décrit les états successifs de la copie du pair p_B. L'exécution de l'opération $\mathsf{ins}(n, \langle v_1, \ldots, v_k\rangle)$ est un raccourci pour l'exécution de k opérations d'insertion qui se succèdent $o_1 \downarrow \cdots \downarrow o_k$ avec $\mathsf{call}(o_i) = \mathsf{ins}(n + i - 1, v_i)$ et $1 \leq i \leq k$. Elle ajoute le n-uplet de valeurs $\langle v_1, \ldots, v_k\rangle$ avant la n-ième valeur de la séquence. Un *dot* $\langle i, n\rangle$ est noté i_n.

Le pair p_A initialise sa séquence (1) et exécute une succession d'opérations qui génère des messages. Il commence par une insertion contiguë de quatre caractères (2). Les positions sont générées de manière à être agrégées. Nous obtenons ainsi le bloc qui contient la chaîne

'*lady*'. Le contexte causal est égal à $\{A_1, \ldots, A_4\}$. p_A exécute ensuite l'opération a_2 qui insère le caractère '*i*' entre '*d*' et '*y*' (3). Aucune position agrégeable à la position de l'un des deux caractères ne peut être générée telle qu'elle soit comprise entre les positions des deux caractères. La position générée a donc pour préfixe la position du caractère '*d*'. L'intégration de cette insertion produit la scission du premier bloc. Le contexte causal est alors égal à $\{A_1, \ldots, A_5\}$. p_A supprime ensuite le caractère '*y*' qui conduit à la suppression du bloc. Le contexte causal demeure inchangé. p_A termine par l'insertion contiguë de deux caractères à la fin de la séquence.

Dotted LogootSplit laisse beaucoup de liberté dans le choix du protocole de synchronisation. Nous illustrons en partie cette liberté par les messages qui sont livrés à p_B. Nous supposons que p_B ne reçoit pas les messages d'insertion générés par p_A du caractère '*i*' et du bloc '*es*'. Nous supposons également que p_A détecte la perte de messages sur le réseau. p_B initialise d'abord sa séquence (1). p_A lui transmet son état complet. p_B intègre l'état de p_A à sa copie (2). Il insère ensuite le caractère '*!*' à la fin de la séquence (3). Une position agrégeable à la position de '*y*' ne peut pas être générée, étant donné que p_B n'est pas le propriétaire du bloc. Il génère une nouvelle position supérieure à celle de '*y*'. Il reçoit ensuite une différence d'états dont l'intégration conduit à la suppression du caractère '*y*' (4). p_A détecte la perte de messages sur le réseau et transmet alors une différence d'états qui provient de la fusion des dernières différences d'états qu'il a transmis (y compris celle qui conduit à la suppression de '*y*'). p_B intègre la différence d'états qu'il reçoit et obtient ainsi la séquence '*ladies!*'.

Le figure 5.12 représente un aperçu du sup-demi-treillis des états de la séquence du pair p_B. Les traits épaissis représentent les transitions d'états du pair p_B des étapes (1) à (4). La fusion de deux états correspond à leur borne supérieure dans le sup-demi treillis.

La séquence obtenue est relativement proche de celle obtenue avec *LogootSplit* (figure 5.7). Cependant, nos positions sont de taille plus faible, étant donné que nous utilisons une composante en moins dans les n-uplets. *Dotted LogootSplit* fait également moins d'hypothèse sur la livraison de messages. Ce qui permet la conception de protocoles de synchronisation plus souples et mieux adaptés aux conditions du réseau ou au type de collaboration rencontré. Dans l'exemple précédent, nous avons combiné la transmission d'états complets avec la transmission de différences d'états irréductibles et la transmission de différences d'états décomposables.

Nous avons implémenté *Dotted LogootSplit*[21] et intégré ce protocole de réplication à l'éditeur collaboratif *MUTE*[22]. Il a été testé dans des conditions réelles à l'aide d'une synchronisation par opérations. Dans ces conditions, il a montré des performances similaires à celles de *LogootSplit*.

21. Le code source est disponible sur *GitHub* : github.com/coast-team/dotted-logootspli
22. coedit.re

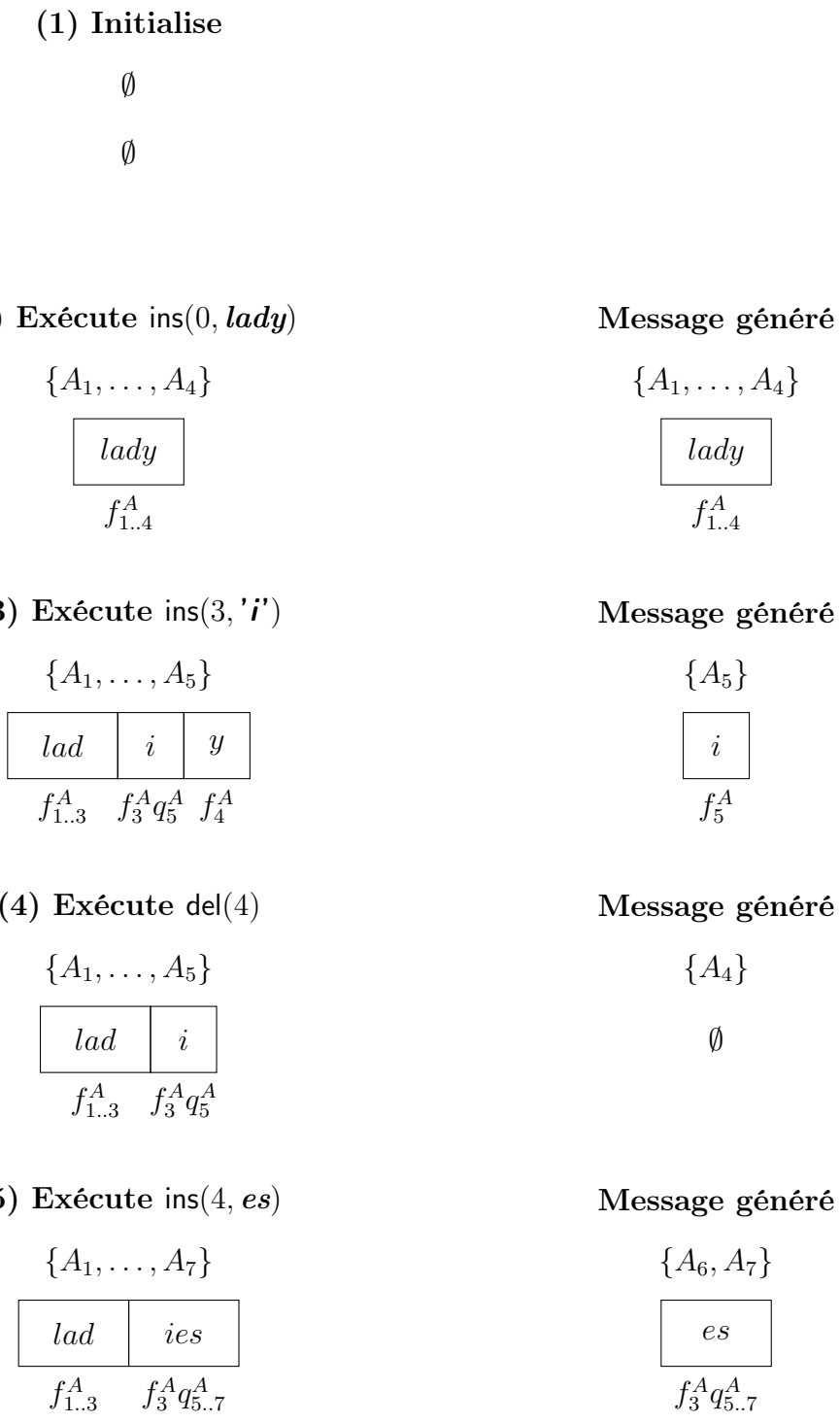


FIGURE 5.10 – États successifs de la copie du pair p_A du scénario de la figure 5.2a. Le contenu répliqué est une séquence *Dotted LogootSplit*.

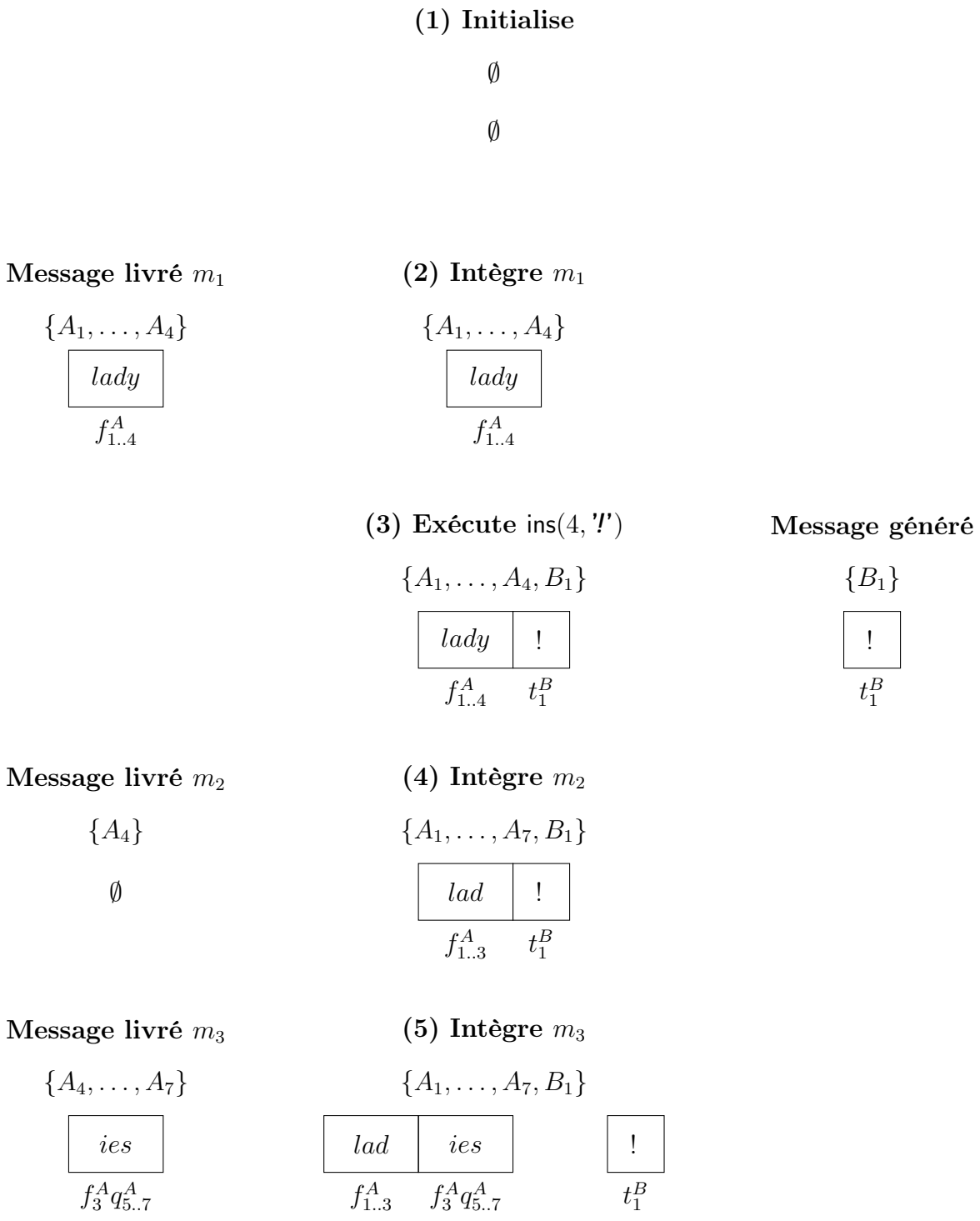


FIGURE 5.11 – États successifs de la copie du pair p_B du scénario de la figure 5.2a. Le contenu répliqué est une séquence *Dotted LogootSplit*.

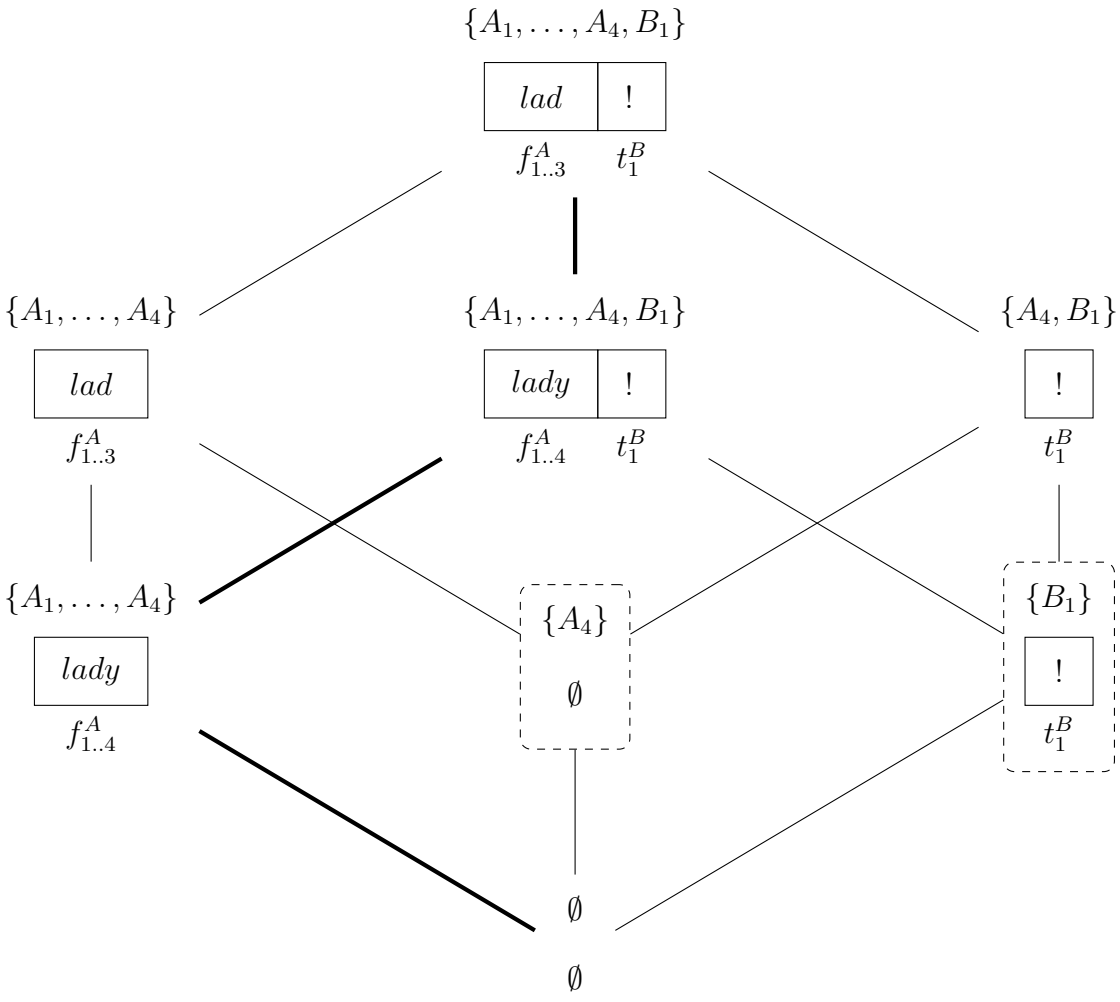


FIGURE 5.12 – Aperçu du diagramme de Hasse du sup-demi-treillis des états d'une séquence *Dotted LogootSplit*. Les états entourés par des tirets sont irréductibles.

5.2.5 Discussion

LogootSplit et *Dotted LogootSplit* agrègent leurs positions pour former des blocs et ainsi réduire leur empreinte mémoire. *LogootSplit* permet au propriétaire d'un bloc d'ajouter à tout moment des valeurs au début et à la fin du bloc. *Dotted LogootSplit* ne permet pas l'ajout de valeurs au début de bloc et restreint l'ajout de valeurs à la fin du bloc. Deux positions agrégeables doivent avoir des nombres séquentiels contiguës. À chaque exécution d'une insertion, le nombre séquentiel est incrémenté. Un pair est donc dans l'impossibilité d'ajouter une valeur au début d'un de ses blocs, étant donné qu'il devrait prendre un nombre séquentiel plus petit que celui de la première valeur du bloc. Il ne peut pas toujours ajouter une valeur en fin de bloc. En effet, s'il a procédé à une insertion à un autre emplacement, son nombre séquentiel devient trop grand pour pouvoir générer une position agrégeable à la position de la dernière valeur du bloc. Cependant, un·e collaborateur·ice à tendance à écrire de manière séquentielle. Nous conjecturons qu'en pratique cette restriction conduit à très peu d'ajouts supplémentaires de blocs. En revanche l'impossibilité d'ajouter des valeurs au début d'un bloc est problématique pour certaines langues écrites de droite à gauche.

5.3 Conclusion

Dotted LogootSplit est le premier CRDT séquence synchronisé par différences d'états. La synchronisation par différences d'états offre une grande flexibilité dans la conception de protocole de synchronisation. Elle permet en particulier à deux pairs de se synchroniser en échangeant des modifications qui peuvent être librement dupliquées et réordonnées. L'intégration d'une modification n'a pas besoin d'être retardée jusqu'à l'intégration d'une ou plusieurs autres modifications. Ce qui devrait permettre d'améliorer le support de l'édition collaborative de texte en simultané. Elle devrait également permettre à des pairs qui se connectent rarement au groupe de se synchroniser plus rapidement en échangeant et en intégrant directement leur état.

Chapitre 6

Conclusion

Sommaire

Les infrastructures pair-à-pair de collaboration et la réplication optimiste des contenus partagés ouvrent la voie vers la conception d'applications de collaboration hautement disponibles, aux latences faibles, qui tolèrent les partitions réseaux, et qui passent à l'échelle. Les protocoles de réplication optimiste proposés pour l'édition collaborative d'un contenu sont généralement vulnérables aux attaques d'un adversaire actif et ne tiennent pas toujours leurs promesses dans des cas d'utilisation exigeants tels que l'édition collaborative de texte en simultané.

Dans ce manuscrit, nous avons proposé deux protocoles qui permettent de répliquer de manière sûre un contenu, et un protocole de réplication optimiste de séquence qui résiste mieux aux aléas du réseau. Ce dernier chapitre est pour nous l'occasion de rappeler nos questions de recherche et d'y répondre. Il synthétise nos contributions et met en évidence leurs limites. Nous proposons plusieurs axes de recherche pour poursuivre nos travaux.

6.1 Convergence sécurisée à l'aide d'un journal tronqué

6.1.1 Résumé des contributions

La communauté scientifique a proposé de nombreux protocoles de réplication optimiste dans le but de concevoir des applications de collaboration hautement disponibles, aux latences faibles, et qui passent à l'échelle. La plupart des protocoles de réplication optimiste ne garantissent pas la convergence des copies du contenu en présence de pairs malintentionnés. Les pairs malintentionnés peuvent en effet produire des équivoques pour compromettre la convergence des copies des pairs honnêtes. Nous nous sommes donc demandés s'il était possible de protéger la convergence des copies des pairs honnêtes tout en préservant les propriétés désirées des protocoles de réplication optimiste.

Nous avons proposé deux protocoles qui protègent la convergence des copies des pairs honnêtes. Notre premier protocole repose sur le maintien d'un journal infalsifiable et répliqué. Chaque pair maintient son propre journal dans lequel il enregistre les messages qu'il transmet aux autres pairs et les messages qu'il reçoit des autres pairs. Notre protocole requiert la préservation de l'intégralité du journal pour maintenir la cohérence des copies et permettre à de nouveaux pairs de joindre la collaboration. La préservation de l'intégralité du journal et sa transmission aux nouveaux pairs engendrent des coûts en mémoire et en communication réseau.

Pour remédier à ces problèmes, nous avons proposé un second protocole qui repose sur le premier et permet la troncature du journal. La troncature du journal est effectuée de manière à protéger la cohérence des copies du contenu partagé. Elle repose sur le concept de *stabilité*. Un message du journal est stable dès lors qu'il devient une dépendance de tout message ajouté ultérieurement dans le journal. Nous avons proposé un algorithme qui permet de déterminer si un message du journal est stable. Cet algorithme prend en compte l'invitation de nouveaux pairs et l'éviction de pairs malintentionnés. Il a une complexité linéaire avec le nombre de messages dans le journal. Pour constituer sa copie du contenu partagé, un nouveau pair récupère un journal tronqué et l'état de la copie associée au journal. Il authentifie l'état de la copie à l'aide du journal tronqué.

Pour montrer la validité de nos protocoles, nous avons proposé un nouveau modèle de cohérence que nous avons nommé View-Fork-Join-Causal Dynamique. Il s'agit d'une restriction du modèle de cohérence View-Fork-Join-Causal (VFJC). Son introduction nous a également amené à proposer le modèle de cohérence Causale Dynamique. Les deux modèles de cohérence que nous avons introduits tiennent compte de la dynamique des groupes et permettent la définition du concept de stabilité. Nous avons également défini le concept de stabilité pour le modèle de cohérence VFJC et le modèle de cohérence Causale.

Nos deux protocoles ne nécessitent pas de coordination entre les pairs. Chaque pair peut à tout moment modifier le contenu répliqué ou tronquer son journal. Par ailleurs, nos protocoles prennent en compte la dynamique des groupes collaboratifs et évincent les pairs qui sont reconnus malintentionnés.

6.1.2 Limites et perspectives

Nous avons apporté des éléments de preuves pour valider nos protocoles. Ces preuves devraient être complétées. Il reste à démontrer que l'exécution de notre protocole à journaux tronqués respecte le modèle de cohérence View-Fork-Join-Causal Dynamique. Nous devrions mener des expérimentations pour évaluer leur capacité à passer à l'échelle.

Pour réduire la complexité algorithmique de la troncature du journal, nous n'identifions pas l'ensemble des messages qui peuvent être supprimés. Nous effectuons une approximation qui nous permet de supprimer un sous-ensemble de ces messages. Nous devrions évaluer par expérimentation si cette approximation n'est pas trop restrictive.

À plus long terme, nous souhaitons explorer d'autres applications du concept de stabilité et améliorer l'utilisabilité de nos protocoles. Nous donnons un aperçu de ces axes de recherche dans les paragraphes suivants.

Nous avons développé le concept de stabilité à travers plusieurs modèles de cohérence et le concept de *stabilité convergente* qui garantit que deux pairs avec le même journal ont un même ensemble de messages stables. Nous avons employé ce concept pour tronquer le journal. Nous pensons que le concept de stabilité peut trouver de nombreuses applications. Il pourrait par exemple être le socle d'un protocole de consensus asynchrones [93]. Dans un protocole de consensus asynchrones, un ensemble de pairs s'accordent sur une valeur unique sans aucune coordination en présence de pairs malintentionnés. Le protocole doit résister aux équivoques des pairs malintentionnés. Les pairs échangent des messages pour trouver un consensus. Un message convergent-stable garantit à un pair que tout sous-ensemble de pairs a connaissance du message et qu'il ne peut prétendre en avoir connaissance à un autre sous-ensemble de pairs. Les pairs peuvent ainsi trouver un consensus de manière déterministe à l'aide de l'ensemble de messages convergents-stables.

Le plus grand problème d'utilisabilité de notre protocole à journaux tronqués réside dans la possibilité de bloquer la troncature du journal. La troncature du journal repose sur le concept de stabilité. La stabilisation de messages exige la participation de l'ensemble des pairs qui ont été invités, à l'exception des pairs évincés. Un pair qui n'exécute pas d'opérations ou qui produit délibérément des messages qui dépendent uniquement de ses anciens messages bloque alors la stabilisation de messages. Pour remédier à ce problème, nous pourrions évincer les pairs qui empêchent la stabilisation de messages.

Les pairs malintentionnés peuvent générer des messages non-linéaires et ainsi produire des embranchements. Nos protocoles rejettent les messages non-linéaires (et par extension les embranchements) qui ne sont pas acceptés par au moins un pair présumé honnête et connu. Les pairs malintentionnés peuvent inviter de nouveaux pairs dans un embranchement. Les pairs honnêtes peuvent donc être amenés à rejeter des invitations. Un nouveau pair peut détecter qu'il a été invité dans un embranchement rejeté par les autres pairs. Il pourrait alors demander à être de nouveau invité par un autre pair.

Deux invitations ne peuvent pas inviter le même pair. Nos protocoles garantissent cette contrainte en générant l'identifiant du pair à partir du message d'invitation. Deux messages d'invitation distincts donnent lieu à deux identifiants de pair distincts. Cette contrainte peut rendre plus difficile l'implémentation des protocoles. Nous avons introduit cette contrainte pour simplifier l'expression de la stabilité. Nous souhaitons étudier les conséquences de la suppression de cette contrainte sur l'expression de la stabilité.

6.2 Séquence répliquée synchronisée par différences

6.2.1 Résumé des contributions

De nombreux protocoles de réplication optimiste de séquences de caractères ont été proposés pour supporter l'édition collaborative de texte en simultané. Pour synchroniser leurs copies du document textuel, les pairs échangent leurs modifications sous la forme d'opérations d'insertion et de suppression. Les opérations doivent être intégrées une seule fois et dans un ordre spécifique. Un ordre d'intégration causal des opérations est généralement supposé.

Les aléas du réseau conduisent au réordonnement et à la perte d'opérations. L'intégration d'opérations peut donc être retardée si l'une de leurs dépendances n'a pas été préalablement intégrée. Ce qui propage des ralentissements dans l'ensemble de la collaboration. Par ailleurs, l'échange et l'intégration des opérations après une longue période de déconnexion d'un·e des collaborateur·ice·s peuvent être coûteuses. Nous nous sommes donc demandé si un protocole de réplication optimiste pour l'édition collaborative de texte en simultané pouvait remédier à ces deux problèmes.

Dans ce manuscrit, nous nous sommes intéressés aux types de données répliquées, communément nommés CRDTs. Nous avons proposé et implémenté un nouveau CRDT séquence nommé *Dotted LogootSplit*. Contrairement aux autres CRDT séquences, il ne retarde pas l'intégration des modifications des pairs. Chaque modification est intégrée dès sa réception. Ce qui évite la propagation de ralentissements, voire l'interruption temporaire, de la collaboration. Il permet également à deux pairs d'échanger directement leur état. Ce qui évite des échanges coûteux de nombreuses modifications lorsqu'un pair se reconnecte après une longue période de déconnexion.

Pour ce faire, *Dotted LogootSplit* synchronise les copies par différences d'états. Une différence d'états peut résumer une seule modification ou plusieurs modifications. Les différences d'états peuvent être librement réordonnées et dupliquées. La synchronisation par différences d'états offre une grande flexibilité dans la conception de protocoles de synchronisation. Notre approche est suffisamment générique pour être adaptée à une famille de CRDTs séquences que nous avons formalisée.

Dotted LogootSplit tire avantage des dernières avancées de l'état de l'art. Il agrége les méta-données associées aux caractères sous la forme de blocs. Ce qui le rend particulièrement adapté à l'édition collaborative de texte.

6.2.2 Limites et perspectives

Dotted LogootSplit a été implémenté et testé dans des conditions réelles. Toutefois, nous n'avons pas conduit des expérimentations pour valider son adéquation à l'édition collaborative de texte en simultané avec un nombre important de collaborateur.ice.s. Pour montrer qu'il se comporte mieux que les protocoles existants face aux aléas du réseau, nous devrions simuler des pertes de messages sur le réseau, des retards de messages, des réordonnements, et des déconnexions temporaires de pairs. Dans ces conditions, nous devrions mesurer les délais entre le moment où une modification est effectuée par un pair et le moment où elle est intégrée par les autres pairs.

À plus long terme, nous souhaitons (i) améliorer la résistance d'une séquence *Dotted LogootSplit* aux attaques d'un adversaire actif, (ii) proposer une stratégie de synchronisation adaptative, et (iii) proposer une stratégie de génération de positions densément ordonnées qui limite les entrelacements de caractères. Nous donnons un aperçu de ces deux axes de recherche dans les paragraphes suivants.

Les protocoles de réplication optimiste sont généralement conçus en supposant l'absence de pairs malintentionnés. Les pairs malintentionnés peuvent produire des équivoques en vue de compromettre la convergence des pairs honnêtes. Certains CRDTs résistent par conception aux équivoques de pairs malintentionnés. Les séquences répliquées associent un identifiant unique à chaque valeur. La synchronisation des copies se base sur ses identifiants uniques. Un pair malintentionné peut produire une équivoque en associant un même identifiant à différentes valeurs. Dans le cas des séquences répliquées à positions densément ordonnées, telle qu'une séquence *Dotted LogootSplit*, un pair malintentionné peut également produire une équivoque en associant à une valeur et son identifiant une position distincte. L'utilisation de nos protocoles à journaux infalsifiables permet de détecter les équivoques. Nous pourrions proposer des mécanismes pour résoudre les conflits de modification introduits par les équivoques. Cependant, nous pensons que ces problématiques devraient être prises en compte dans la conception des CRDTs. Elles devraient en particulier être prises en compte dans la conception de CRDTs synchronisés par différences d'états puisque l'utilisation d'un journal infalsifiable annule la plupart de leurs avantages. En effet, l'utilisation d'un journal infalsifiable nécessite la réintroduction de dépendances causales entre les modifications.

Dotted LogootSplit offre une grande flexibilité dans la conception du protocole de synchronisation. Il peut utiliser un protocole de synchronisation par opérations, un protocole de synchronisation par états complets, ou l'un des protocoles de synchronisation par différences d'états proposés pour les CRDTs. Nous pourrions évaluer les performances de différents protocoles de synchronisation en fonction des paramètres que sont la taille du contenu, le nombre de modifications effectuées depuis la dernière synchronisation, le mode de collaboration utilisé (en simultané ou en différé), la qualité du réseau. À partir de cette évaluation, nous pourrions proposer une stratégie adaptative qui choisit le protocole de synchronisation de manière dynamique au cours de la collaboration.

La modification parallèle d'un contenu partagé rend difficile la définition de l'intention des opérations que les collaborateur·ice·s exécutent. Concernant l'édition collaborative de texte, il est communément accepté qu'une insertion à un indice donné traduit l'intention de vouloir insérer un caractère entre les caractères qui entourent cet indice. Kleppmann et al. [87] ont mis en évidence l'importance de rejeter les entrelacements de caractères insérés en parallèle. *Dotted Logoot* peut produire des entrelacements. Toutefois, il pourrait tirer avantage de l'agrégeabilité de ses positions pour rejeter certains entrelacements de caractères. Lorsqu'un pair insère des caractères à un même emplacement les uns à la suite des autres, les caractères sont associés à des positions agrégeables. Les caractères sont groupés au sein d'un bloc. Le générateur de position pourrait garantir que tout caractère inséré en parallèle de ces caractères est placé avant ou après ces derniers.

6.3 Notes finales

Les infrastructures pair-à-pair de collaboration et la réplication optimiste des contenus partagés offrent des défis difficiles et stimulants. Ils nous poussent à remettre en question des pratiques établies et à trouver de nouveaux compromis. La suppression du serveur et la promotion des collaborateur·ice·s en pairs responsables de la réplication du contenu partagé permettent de dépasser les limites des architectures centralisées, mais apportent aussi de nouvelles problématiques.

Les infrastructures pair-à-pair de collaboration présentent une surface d'attaque plus large qu'une architecture centralisée, étant donnée la responsabilité conférée aux pairs. Les protocoles de réplication optimiste répondent déjà à de nombreuses problématiques difficiles. Leur sécurisation est une problématique qui apporte ses propres difficultés. Nous comprenons aujourd'hui davantage les problèmes que soulève la conception de protocoles de réplication optimiste et la manière d'y répondre. Les protocoles de réplication optimiste ont fait leurs preuves et nous pensons qu'il est désormais temps de prendre en compte les problématiques de sécurité dès leur conception.

Bibliographie

[1] Stephen Mogan et Weigang Wang. « The Impact of Web 2.0 Developments on Real-Time Groupware ». In : *Proceedings of the 2nd IEEE International Conference on Social Computing, SocialCom 2010*. Sous la dir. d'Ahmed K. Elmagarmid et Divyakant Agrawal. Août 2010, p. 534-539. doi : 10.1109/SocialCom.2010.84.

[2] Clarence A Ellis, Simon J Gibbs et Gail Rein. « Groupware : Some Issues and Experiences ». In : *Commununications of the ACM* 34.1 (jan. 1991), p. 39-58. doi : 10.1145/99977.99987.

[3] Kevin Dooley. *Designing Large Scale Lans : Help for Network Designers*. O'Reilly Media, Inc., nov. 2001. isbn : 978-0596001506.

[4] Claudia-Lavinia Ignat, Gérald Oster, Olivia Fox, Valerie L. Shalin et François Charoy. « How Do User Groups Cope with Delay in Real-Time Collaborative Note Taking ». In : *Proceedings of the 14th European Conference on Computer Supported Cooperative Work, ECSCW 2015*. Sept. 2015, p. 223-242. doi : 10.1007/978-3-319-20499-4_12.

[5] Claudia-Lavinia Ignat, Gérald Oster, Meagan Newman, Valerie Shalin et François Charoy. « Studying the Effect of Delay on Group Performance in Collaborative Editing ». In : *Cooperative Design, Visualization, and Engineering - 11th International Conference, CDVE 2014*. Sous la dir. d'Yuhua Luo. T. 8683. Lecture Notes in Computer Science. Springer, sept. 2014, p. 191-198. doi : 10.1007/978-3-319-10831-5_29.

[6] M. Claude Jard. « Un langage de composition des techniques de sécurité pour préserver la vie privée dans le nuage ». Thèse de doct. Université de Nice, 2016.

[7] Paul Dourish. « The Parting of the Ways : Divergence, Data Management and Collaborative Work ». In : *Proceedings of the 4th European Conference on Computer Supported Cooperative Work, ECSCW 1995*. Sept. 1995, p. 215-230. doi : 10.1007/978-94-011-0349-7_14.

[8] Juan Benet. « IPFS - Content Addressed, Versioned, P2P File System ». In : *CoRR* abs/1407.3561 (2014). arXiv : 1407.3561.

[9] Gavin Wood. « Ethereum : A secure decentralised generalised transaction ledger ». In : *Ethereum project yellow paper* 151.2014 (2014), p. 1-32.

[10] Essam MANSOUR, Andrei Vlad SAMBRA, Sandro HAWKE, Maged ZEREBA, Sarven CAPADISLI, Abdurrahman GHANEM, Ashraf ABOULNAGA et 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*. 2016, p. 223-226.

[11] Rodrigo RODRIGUES et Peter DRUSCHEL. « Peer-to-peer systems ». In : *Communications of the ACM* 53.10 (oct. 2010), p. 72-82. DOI : 10.1145/1831407.1831427.

[12] Yasushi SAITO et Marc SHAPIRO. « Optimistic Replication ». In : *ACM Computing Surveys* 37.1 (mars 2005), p. 42-81. DOI : 10.1145/1057977.1057980.

[13] Carlos BAQUERO, Paulo Sérgio ALMEIDA et Ali SHOKER. « Pure Operation-Based Replicated Data Types ». In : *CoRR* abs/1710.04469 (2017). arXiv : 1710.04469. URL : http://arxiv.org/abs/1710.04469.

[14] Jinyuan LI, Maxwell N. KROHN, David MAZIÈRES et Dennis E. SHASHA. « Secure Untrusted Data Repository (SUNDR) ». In : *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*. Déc. 2004, p. 121-136.

[15] Prince MAHAJAN, Srinath SETTY, Sangmin LEE, Allen CLEMENT, Lorenzo ALVISI, Mike DAHLIN et Michael WALFISH. « Depot : Cloud Storage with Minimal Trust ». In : *ACM Transactions on Computer Systems (TOCS)* 29.4 (2011), 12 :1-12 :38. DOI : 10.1145/2063509.2063512.

[16] Hien Thi Thu TRUONG, Claudia-Lavinia IGNAT et Pascal MOLLI. « Authenticating Operation-based History in Collaborative Systems ». In : *Proceedings of the 17th ACM International Conference on Supporting Group Work (GROUP 2012)*. ACM, 2012, p. 131-140. DOI : 10.1145/2389176.2389197.

[17] Ariel J. FELDMAN, William P. ZELLER, Michael J. FREEDMAN et Edward W. FELTEN. « SPORC : Group Collaboration Using Untrusted Cloud Resources ». In : *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010)*. USENIX Association, 2010, p. 337-350. ISBN : 978-1-931971-79-9.

[18] Paulo Sérgio ALMEIDA, Ali SHOKER et Carlos BAQUERO. « Delta State Replicated Data Types ». In : *Journal of Parallel and Distributed Computing* 111 (2018), p. 162-173. URL : http://arxiv.org/abs/1603.01529.

[19] Quang-Vinh DANG et Claudia-Lavinia IGNAT. « Performance of real-time collaborative editors at large scale : User perspective ». In : *Proceedings of the 15th International Conference on IFIP Networking, Networking 2016 and Workshops*. IEEE Computer Society, mai 2016, p. 548-553. DOI : 10.1109/IFIPNetworking.2016.7497258.

[20] Mehdi AHMED-NACER, Claudia-Lavinia IGNAT, Gérald OSTER, Hyun-Gul ROH et Pascal URSO. « Evaluating CRDTs for real-time document editing ». In : *Proceedings of the 11th ACM Symposium on Document Engineering, DocEng 2011*. Sous la dir. de Matthew R. B. HARDY et Frank Wm. TOMPA. ACM, sept. 2011, p. 103-112. DOI : 10.1145/2034691.2034717.

[21] Gérald Oster, Pascal Urso, Pascal Molli et Abdessamad Imine. « Data consistency for P2P collaborative editing ». In : *Proceedings of the 2006 ACM Conference on Computer Supported Cooperative Work, CSCW 2006*. ACM, nov. 2006, p. 259-268. doi : 10.1145/1180875.1180916.

[22] Nuno M. Preguiça, Joan Manuel Marquès, Marc Shapiro et Mihai Letia. « A Commutative Replicated Data Type for Cooperative Editing ». In : *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems, (ICDCS 2009*. Juin 2009, p. 395-403. doi : 10.1109/ICDCS.2009.20.

[23] Stéphane Weiss, Pascal Urso et Pascal Molli. « Logoot-Undo : Distributed Collaborative Editing System on P2P Networks ». In : *IEEE Transactions on Parallel and Distributed Systems* 21.8 (2010), p. 1162-1174. doi : 10.1109/TPDS.2009.173.

[24] Hyun-Gul Roh, Myeongjae Jeon, Jinsoo Kim et Joonwon Lee. « Replicated abstract data types : Building blocks for collaborative applications ». In : *Journal of Parallel and Distributed Computing* 71.3 (2011), p. 354-368. doi : 10.1016/j.jpdc.2010.12.006.

[25] Luc André, Stéphane Martin, Gérald Oster et Claudia-Lavinia Ignat. « Supporting adaptable granularity of changes for massive-scale collaborative editing ». In : *Proceedings of the 9th International Conference on Collaborative Computing : Networking, Applications and Worksharing (Collaboratecom)*. Oct. 2013, p. 50-59. doi : 10.4108/icst.collaboratecom.2013.254123.

[26] Petru Nicolaescu, Kevin Jahns, Michael Derntl et Ralf Klamma. « Yjs : A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types ». In : *Proceedings of the 15th International Conference on Engineering the Web in the Big Data Era, ICWE 2015*. Sous la dir. de Philipp Cimiano, Flavius Frasincar, Geert-Jan Houben et Daniel Schwabe. T. 9114. Lecture Notes in Computer Science. Springer, juin 2015, p. 675-678. doi : 10.1007/978-3-319-19890-3_55.

[27] Loïck Briot, Pascal Urso et Marc Shapiro. « High Responsiveness for Group Editing CRDTs ». In : *Proceedings of the 19th International Conference on Supporting Group Work*. Nov. 2016, p. 51-60. doi : 10.1145/2957276.2957300.

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

[29] Marc Shapiro, Nuno Preguiça, Carlos Baquero et Marek Zawirski. *A comprehensive study of convergent and commutative replicated data types*. Research Report RR-7506. Jan. 2011.

[30] Ravi Prakash, Michel Raynal et Mukesh Singhal. « An Adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments ». In : *Journal of Parallel and Distributed Computing* 41.2 (1997), p. 190-204. doi : 10.1006/jpdc.1996.1300.

[31] Lorenzo Alvisi, Natacha Crooks et Syed Akbar Mehdi. « Writes : the dirty secret of causal consistency ». In : *IEEE Data Engineering Bulletin* 40.4 (2017), p. 15-25. URL : http://sites.computer.org/debull/A17dec/p15.pdf.

[32] Victorien Elvinger, Gérald Oster et François Charoy. « Prunable Authenticated Log and Authenticable Snapshot in Distributed Collaborative Systems ». In : *Proceedings of the 4th IEEE International Conference on Collaboration and Internet Computing, CIC 2018*. Oct. 2018, p. 156-165. DOI : 10.1109/CIC.2018.00031.

[33] Matthieu Nicolas, Victorien Elvinger, Gérald Oster, Claudia-Lavinia Ignat et François Charoy. « MUTE : A Peer-to-Peer Web-based Real-time Collaborative Editor ». In : *Proceedings of the 15th European Conference on Computer Supported Cooperative Work - Panels, Demos and Posters, ECSCW 2017*. Août 2017. DOI : 10.18420/ecscw2017_p5.

[34] Weihai Yu, Victorien Elvinger et Claudia-Lavinia Ignat. « A Generic Undo Support for State-Based CRDTs ». In : *23rd International Conference on Principles of Distributed Systems, (OPODIS 2019)*. Sous la dir. de Pascal Felber, Roy Friedman, Seth Gilbert et Avery Miller. T. 153. LIPIcs. Déc. 2019, 14 :1-14 :17. DOI : 10.4230/LIPIcs.OPODIS.2019.14.

[35] Arnon Rotem-Gal-Oz. « Fallacies of distributed computing explained ». In : *RGO Architects* 20 (2006).

[36] Eric A. Brewer. « Towards robust distributed systems (abstract) ». In : *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC 2000)*. Juill. 2000, p. 7. DOI : 10.1145/343477.343502.

[37] Seth Gilbert et Nancy A. Lynch. « Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services ». In : *SIGACT News* 33.2 (juin 2002), p. 51-59. DOI : 10.1145/564585.564601.

[38] Usman W Chohan. « The Double Spending Problem and Cryptocurrencies ». In : *Available at SSRN 3090174* (2017).

[39] Doug Terry. « Replicated data consistency explained through baseball ». In : *Communications of the ACM* 56.12 (2013), p. 82-89.

[40] Joseph M. Hellerstein et Peter Alvaro. « Keeping CALM : When Distributed Consistency is Easy ». In : *CoRR* abs/1901.01930 (2019).

[41] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall et Werner Vogels. « Dynamo : amazon's highly available key-value store ». In : *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007)*. Oct. 2007, p. 205-220. DOI : 10.1145/1294261.1294281.

[42] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer et Brent B. Welch. « Session Guarantees for Weakly Consistent Replicated Data ». In : *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems (PDIS 94)*. Sept. 1994, p. 140-149. DOI : 10.1109/PDIS.1994.331722.

[43] Werner VOGELS. « Eventually consistent ». In : *Commun. ACM* 52.1 (2009), p. 40-44. DOI : 10.1145/1435417.1435432.

[44] Peter BAILIS, Alan FEKETE, Michael J. FRANKLIN, Ali GHODSI, Joseph M. HELLERSTEIN et Ion STOICA. « Coordination Avoidance in Database Systems ». In : *PVLDB* 8.3 (2014), p. 185-196. DOI : 10.14778/2735508.2735509.

[45] Bowen ALPERN et Fred B. SCHNEIDER. « Defining Liveness ». In : *Information Processing Letters* 21.4 (oct. 1985), p. 181-185. DOI : 10.1016/0020-0190(85)90056-0.

[46] Douglas B. TERRY, Marvin THEIMER, Karin PETERSEN, Alan J. DEMERS, Mike SPREITZER et Carl HAUSER. « Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System ». In : *Proceedings of the 15th ACM Symposium on Operating System Principles, (SOSP 1995)*. Déc. 1995, p. 172-183. DOI : 10.1145/224056.224070.

[47] Prince MAHAJAN, Lorenzo ALVISI et Mike DAHLIN. *Consistency, Availability, and Convergence*. Rapp. tech. TR-11-22. Department of Computer Science, The University of Texas at Austin, 2011.

[48] Rachid GUERRAOUI, Matej PAVLOVIC et Dragos-Adrian SEREDINSCHI. « Trade-offs in Replicated Systems ». In : *IEEE Data Engineering Bulletin* 39.1 (mars 2016), p. 14-26. URL : http://sites.computer.org/debull/A16mar/p14.pdf.

[49] Nancy A. LYNCH. « Distributed Algorithms ». In : The Morgan Kaufmann Series in Data Management Systems, 1996. Chap. 14, p. 457-471. ISBN : 1558603484.

[50] Leslie LAMPORT, Robert E. SHOSTAK et Marshall C. PEASE. « The Byzantine Generals Problem ». In : *ACM Transactions on Programming Languages and Systems* 4.3 (1982), p. 382-401. DOI : 10.1145/357172.357176.

[51] Sebastian BURCKHARDT. « Principles of Eventual Consistency ». In : *Foundations and Trends in Programming Languages* 1.1-2 (2014), p. 1-150. DOI : 10.1561/2500000011.

[52] T Lockman GREENOUGH. « representation and enumeration of interval orders and semiorders ». In : (1976).

[53] Paolo VIOTTI et Marko VUKOLIĆ. « Consistency in Non-Transactional Distributed Storage Systems ». In : *ACM Computing Surveys (CSUR)* 49.1 (2016), 19 :1-19 :34. DOI : 10.1145/2926965.

[54] Leslie LAMPORT. « Proving the Correctness of Multiprocess Programs ». In : *IEEE Transactions of Software Engineering* 3.2 (1977), p. 125-143. DOI : 10.1109/TSE.1977.229904.

[55] Leslie LAMPORT. « Time, Clocks, and the Ordering of Events in a Distributed System ». In : *Communications of the ACM* 21.7 (1978), p. 558-565. DOI : 10.1145/359545.359563.

[56] Phillip W. Hutto et Mustaque Ahamad. « Slow Memory : Weakening Consistency to Enchance Concurrency in Distributed Shared Memories ». In : *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS 1990)*. Mai 1990, p. 302-309. doi : 10.1109/ICDCS.1990.89297.

[57] Saul Eduardo Pomares Hernández, Jean Fanchon et Khalil Drira. « The Immediate Dependency Relation : an Optimal Way to Ensure Causal Group Communication ». In : *Annual Review of Scalable Computing, Editions World Scientific, Series on Scalable Computing*. T. 6. 2003, p. 60-79.

[58] Saúl Eduardo Pomares Hernández. « The Minimal Dependency Relation for Causal Event Ordering in Distributed Computing ». In : *Applied Mathematics & Information Sciences* 9.1 (2015), pp.57-61. doi : 10.12785/amis/090108.

[59] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli et Phillip W. Hutto. « Causal Memory : Definitions, Implementation, and Programming ». In : *Distributed Computing* 9.1 (1995), p. 37-49. doi : 10.1007/BF01784241.

[60] Colin J Fidge. « Timestamps in message-passing systems that preserve the partial ordering ». In : (1987).

[61] Prince Mahajan, Sangmin Lee, An Zheng, Lorenzo Alvisi et Mike Dahlin. *ASTRO : Autonomous and Trustworthy Data Sharing*. Rapp. tech. TR-09-29. Department of Computer Science, The University of Texas at Austin, 2008.

[62] Gérald Oster, Pascal Urso, Pascal Molli et Abdessamad Imine. « Proving correctness of transformation functions in collaborative editing systems ». In : (2005).

[63] Nuno Preguiça, Carlos Baquero et Marc Shapiro. *Conflict-free Replicated Data Types (CRDTs)*. Mai 2018. arXiv : 1805.06358 [cs.DC].

[64] Martin Wirsing. « Algebraic Specification ». In : *Handbook of Theoretical Computer Science, Volume B : Formal Models and Semantics*. Sous la dir. de Jan van Leeuwen. Elsevier et MIT Press, 1990, p. 675-788. doi : 10.1016/b978-0-444-88074-1.50018-4.

[65] Weihai Yu et Sigbjørn Rostad. « A low-cost set CRDT based on causal lengths ». In : *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2020*. Avr. 2020, 5 :1-5 :6.

[66] Carlos Baquero, Paulo Sérgio Almeida et Ali Shoker. « Making Operation-Based CRDTs Operation-Based ». In : *Distributed Applications and Interoperable Systems - 14th IFIP WG 6.1 International Conference, DAIS 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques*. Lecture Notes in Computer Science. Springer, juin 2014, p. 126-140. doi : 10.1007/978-3-662-43352-2_11.

[67] Brian A. Davey et Hilary A. Priestley. *Introduction to Lattices and Order, Second Edition*. Cambridge University Press, 2002. isbn : 978-0-521-78451-1. doi : 10.1017/CBO9780511809088.

[68] Douglas Stott Parker Jr., Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David A. Edwards, Stephen Kiser et Charles S. Kline. « Detection of Mutual Inconsistency in Distributed Systems ». In : *IEEE Transactions on Software Engineering* 9.3 (1983), p. 240-247. doi : 10.1109/TSE.1983.236733.

[69] Friedemann Mattern. « Virtual time and global states of distributed systems ». In : *Proceedings of International Workshop on Parallel Algorithms*. Oct. 1988, p. 215-230.

[70] Vitor Enes, Paulo Sérgio Almeida, Carlos Baquero et João Leitão. « Efficient Synchronization of State-based CRDTs ». In : *CoRR* abs/1803.02750 (2018). url : https://arxiv.org/abs/1803.02750.

[71] Kenneth P. Birman, André Schiper et Pat Stephenson. « Lightweigt Causal and Atomic Group Multicast ». In : *ACM Transactions on Computer Systems* 9.3 (août 1991), p. 272-314. doi : 10.1145/128738.128742.

[72] Michael Engen et Vincent Vatter. *Containing all permutations*. 2018. arXiv : 1810.08252 [math.CO].

[73] Leonard Adleman. « Short permutation strings ». In : *Discrete Mathematics* 10.2 (1974), p. 197-200.

[74] P. J. Koutas et T. C. Hu. « Shortest string containing all permutations ». In : *Discrete Mathematics* 11.2 (1975), p. 125-132. doi : 10.1016/0012-365X(75)90004-7.

[75] Giulia Galbiati et Franco P Preparata. « On permutation-embedding sequences ». In : *SIAM Journal on Applied Mathematics* 30.3 (1976), p. 421-423.

[76] Eugen Zălinescu. « Shorter strings containing all k-element permutations ». In : *Information processing letters* 111.12 (2011), p. 605-608.

[77] Sasa Radomirovic. « A construction of short sequences containing all permutations of a set as subsequences ». In : *the electronic journal of combinatorics* 19.4 (2012), P31.

[78] Richard A. Golding. « A Weak-Consistency Architecture for Distributed Information Services ». In : *Computing Systems* 5.4 (1992), p. 379-405.

[79] Sunil K. Sarin et Nancy A. Lynch. « Discarding obsolete information in a replicated database system ». In : *IEEE Transactions on Software Engineering* 1 (1987), p. 39-47.

[80] Stephan A. Kollmann, Martin Kleppmann et Alastair R. Beresford. « Snapdoc : Authenticated snapshots with history privacy in peer-to-peer collaborative editing ». In : *PoPETs* 2019.3 (2019), p. 210-232. doi : 10.2478/popets-2019-0044.

[81] Ralph Charles Merkle. « Secrecy, Authentication, and Public Key Systems ». AAI8001972. Thèse de doct. Stanford, CA, USA : Stanford University, 1979.

[82] Brice Nédelec, Pascal Molli et Achour Mostefaoui. « CRATE : Writing Stories Together with our Browsers ». In : *Proceedings of the 25th International Conference Companion on World Wide Web, WWW 2016*. Sous la dir. de Jacqueline Bourdeau, Jim Hendler, Roger Nkambou, Ian Horrocks et Ben Y. Zhao. ACM, avr. 2016, p. 231-234. DOI : 10.1145/2872518.2890539.

[83] Stéphane Weiss, Pascal Urso et Pascal Molli. « Logoot : A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks ». In : *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems, ICDCS 2009*. Juin 2009, p. 404-412. DOI : 10.1109/ICDCS.2009.75.

[84] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang et David Chen. « Achieving Convergence, Causality Preservation, and Intention Preservation in Real-time Cooperative Editing Systems ». In : *ACM Transactions on Computer-Human Interaction* 5.1 (mars 1998), p. 63-108. DOI : 10.1145/274444.274447.

[85] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang et Marek Zawirski. « Specification and Complexity of Collaborative Text Editing ». In : *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC)*. Juill. 2016, p. 259-268. DOI : 10.1145/2933057.2933090.

[86] Hengfeng Wei, Yu Huang et Jian Lu. « Specification and Implementation of Replicated List : The Jupiter Protocol Revisited ». In : *Proceedings of the 22nd International Conference on Principles of Distributed Systems, OPODIS 2018*. Sous la dir. de Jiannong Cao, Faith Ellen, Luis Rodrigues et Bernardo Ferreira. T. 125. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, déc. 2018, 12 :1-12 :16. DOI : 10.4230/LIPIcs.OPODIS.2018.12.

[87] Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan et Alastair R. Beresford. « Interleaving anomalies in collaborative text editors ». In : *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2019*. ACM, mars 2019, 6 :1-6 :7.

[88] Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan et Alastair R. Beresford. « OpSets : Sequential Specifications for Replicated Datatypes (Extended Version) ». In : *CoRR* abs/1805.04263 (2018). arXiv : 1805.04263. URL : http://arxiv.org/abs/1805.04263.

[89] Stéphane Weiss, Pascal Urso et Pascal Molli. « Wooki : A P2P Wiki-Based Collaborative Writing Tool ». In : *Proceedings of the 8th International Conference on Web Information Systems Engineering, WISE 2007*. Sous la dir. de Boualem Benatallah, Fabio Casati, Dimitrios Georgakopoulos, Claudio Bartolini, Wasim Sadiq et Claude Godart. T. 4831. Lecture Notes in Computer Science. Springer, déc. 2007, p. 503-512. DOI : 10.1007/978-3-540-76993-4_42.

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

[91] Weihai YU. « A string-wise CRDT for group editing ». In : *Proceedings of the 2012 ACM International Conference on Support Group Work, GROUP 2012*. Oct. 2012, p. 141-144. DOI : 10.1145/2389176.2389198.

[92] Matthieu NICOLAS, Gérald OSTER et Olivier PERRIN. « Efficient renaming in sequence CRDTs ». In : *7th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2020, Heraklion, Greece, April 27, 2020*. Sous la dir. d'Alan FEKETE et Martin KLEPPMANN. ACM, avr. 2020, 9 :1-9 :8. DOI : 10.1145/3380787.3393682.

[93] Gabriel BRACHA et Sam TOUEG. « Asynchronous Consensus and Broadcast Protocols ». In : *ournal of the ACM* 32.4 (1985), p. 824-840. DOI : 10.1145/4221.214134.

Index

Colophon

Ce manuscrit a été écrit à l'aide de LaTeX. Il utilise la classe de mise en forme *thesul*. Les figures ont été réalisés à l'aide de *Tikz*, Inkscape (inkscape.org), et draw.io. Les figures sont licenciées sous Creative Common Attribution 4.0 International (CC BY 4.0).

Les programmes développés et présentés dans ce manuscrit sont sous licence GNU Affero General Public License version 3.0 (AGPLv3) ou Mozilla Public License version 2.0 (MPLv2). Ils sont disponibles sur *Github* (github.com/coast-team).

Réplication sécurisée dans les infrastructures pair-à-pair de collaboration

Résumé

Une application de collaboration permet à plusieurs individus de coéditer un contenu. Les infrastructures pair-à-pair de collaboration visent à la conception d'applications hautement disponibles, aux latences faibles, qui tolèrent les partitions réseaux, et qui passent à l'échelle. Chaque pair (individu) modifie sa propre copie du contenu. La modification concurrente des copies conduit à leur divergence. Les protocoles de réplication sont responsables de la convergence des copies.

Ces protocoles supposent l'absence de pairs malintentionnés qui compromettent la convergence des copies. Pouvons-nous protéger la convergence des copies et préserver les propriétés des infrastructures pair-à-pair ? Nous proposons deux protocoles qui protègent la convergence des copies. Le premier protocole maintient un journal répliqué et infalsifiable qui enregistre les modifications du contenu. Les pairs conservent l'intégralité du journal pour déjouer les attaques des pairs malintentionnés et pour le transmettre à ceux qui rejoignent la collaboration. Le second protocole permet aux pairs de tronquer leur journal. La troncature du journal repose sur le concept de *Stabilité*. Une modification devient stable lorsque toute modification intégrée dans le journal dépend d'elle. Pour rejoindre la collaboration, un pair récupère une copie et un journal tronqué. Il vérifie l'authenticité de la copie à partir du journal tronqué.

Un type de données répliquées (*CRDT*) encapsule un protocole de réplication. Les *CRDTs* séquences supposent généralement un ordre d'intégration causal des modifications du contenu. Le retard d'une modification propage des ralentissements dans l'ensemble du système. La connexion d'un pair peut engendrer l'intégration de nombreuses modifications. Pouvons-nous éliminer ces ralentissements et ces intégrations coûteuses ? Nous formalisons une famille de *CRDTs* séquences et nous proposons une approche qui permet leur synchronisation par différences d'états. Les différences d'états peuvent être intégrées dans un ordre arbitraire et résumer plusieurs modifications. Nous proposons un *CRDT* séquence qui tire avantage de notre approche.

Abstract

A collaborative application allows multiple users to edit a shared content. Collaborative peer-to-peer environments aim to design applications with desirable properties: high-availability, low-latency, fault-tolerance, and scalability. Every peer (user) modifies her own copy of the content. Concurrent modifications of the copies lead to their divergence. Replication protocols are responsible for the convergence of copies.

These protocols assume the absence of malicious peers that compromise the convergence of copies. Can we secure the convergence of copies and preserve the properties of collaborative peer-to-peer environment? We propose two protocols to secure convergence. The first protocol maintains a replicated and unforgeable log that stores the modifications of the content. Peers preserve the full log in order to foil the attacks of malicious peers and to transmit it to new peers. The second protocol enables the peers to truncate their log. The log truncation relies on the concept of *Stability*. A modification becomes stable once every modification added in the log depends on it. To join the collaboration, a peer retrieves a copy of the content and a truncated log used for checking the copy authenticity.

A Conflict-free Replicated Data Type (CRDT) encapsulates a replication protocol. Sequence *CRDTs* generally assume the integration of content modifications in a causal order. A delayed integration propagates slowness across the system. The connection of a peer may lead to the integration of multiple modifications. Can we remove these delays and these costly integrations? We formalize a family of sequence *CRDTs* and we propose their synchronization using delta states. Delta states can be integrated in any order. They are able to summarize several modifications. We propose a sequence *CRDT* that takes advantage of our approach.