Spécification des objets partagés dans les systèmes répartis sans-attente

Matthieu Perrin

UNIVERSITE BRETAGNE LOIRE

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Matthieu PERRIN

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

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

Discipline : Informatique et applications, section CNU 27
Unité de recherche : Laboratoire d'informatique de Nantes-Atlantique (LINA)

Soutenue le 07/06/2016

Spécification des objets partagés dans les systèmes répartis sans-attente

JURY

Président :	**M. Jean-Marc MENAUD**, Professeur, École des Mines de Nantes
Rapporteurs :	**M. Luc BOUGÉ**, Professeur, École Normale Supérieure de Rennes **M^me^ Maria POTOP-BUTUCARU**, Professeur, Université Pierre et Marie Curie, Paris
Examinateur :	**M^me^ Alessia MILANI**, Maître de conférences, ENSEIRB-MATMECA, Bordeaux
Directeur de thèse :	**M. Claude JARD**, Professeur, Université de Nantes
Co-directeur de thèse :	**M. Achour MOSTÉFAOUI**, Professeur, Université de Nantes

Remerciements

Je tiens tout d'abord à remercier les membres du jury d'avoir accepté de porter un regard extérieur sur mon travail, et en particulier les rapporteurs Luc Bougé et Maria Potop-Butucaru, dont les remarques nombreuses et constructives ont beaucoup participé à l'amélioration de ce manuscrit de thèse. Je tiens également à remercier plus spécifiquement Maria, ainsi qu'Olivier-Henri Roux, deuxième membre extérieur de mon comité de suivi de thèse, pour leurs conseils annuels, et Luc pour l'idée d'un co-encadrement par Claude Jard et Achour Mostéfaoui.

Claude et Achour, justement, m'ont apporté un soutien indispensable pendant toute la préparation de ma thèse, au cours de laquelle ils m'ont toujours considéré comme un collègue chercheur à part entière. Je tiens à les remercier chaleureusement pour leur confiance sans cesse renouvelée et la liberté qu'ils m'ont laissée pour développer mes propres idées. Les nombreuses discussions sur tous les aspects des systèmes répartis (et sur beaucoup d'autres choses), en particulier avec Achour, ont été des moments particulièrement enrichissants et productifs.

Outre Claude et Achour, j'ai eu la chance de rencontrer et de travailler avec de nombreuses personnes pendant ma thèse. Je tiens à remercier en particulier Davide Frey, Damien Maussion, Matoula Petrolia, Pierre-Louis Roman, Olivier Ruas, Hala Skaff-Molli et François Taïani. Plus généralement, je remercie tous les membres de mes deux équipes, GDD et AeLoS, pour leur accueil, et en particulier Pascal Molli dont l'imagination débordante est une assurance contre le syndrome de la page blanche. Les interactions avec eux m'ont aidé à garder à l'esprit les applications pratiques de mes recherches. Merci également aux membres des projets SocioPlug, Descent et Displexity pour les réunions de travail toujours passionnantes auxquelles j'ai été convié, ainsi que pour le financement de plusieurs missions.

Je tiens également à remercier le personnel administratif et technique du laboratoire, du département d'enseignement et de l'Université. Leur rôle primordial dans la vie de la recherche et de l'enseignement supérieur n'est pas assez souligné. Une pensée spéciale revient à Sylvie Lartault et Françoise Le Fichant qui m'ont apporté un soutien précieux à un moment particulièrement difficile de ma thèse.

Je remercie également mes amis et ma famille pour le soutien moral qu'ils m'ont apporté pendant toutes ces années, et en particulier tous les doctorants du laboratoire. Concernant plus précisément ce document, je remercie mes parents pour l'attention particulière qu'ils ont apportée à la relecture de ma thèse, ainsi que Kajal Bangera pour sa relecture du résumé en anglais.

Table des matières

1

Introduction

Sommaire

Préambule

Bob était tout songeur en s'engouffrant dans la bouche de métro. Voilà deux jours qu'il n'avait pas parlé à Alice. Sa colère s'était peu à peu transformée en amertume, puis en culpabilité. Il comprenait maintenant que sa réaction initiale avait été exagérée, et désirait plus que tout faire la paix avec son amie d'enfance mais ne savait pas comment l'aborder. Aussi quel ne fut pas son soulagement lorsqu'il reçut un message d'elle l'invitant à sortir boire un café. Il s'empressa de répondre alors que la rame s'avançait dans le tunnel sombre. Hélas, au plus profond de la ville, le réseau défaillant ne put transmettre son « Évidemment » libérateur.

Alice aimait Bob comme une sœur, malgré l'anxiété chronique et le caractère ombrageux de celui qu'elle voyait comme son plus ancien ami. Deux jours plus tôt, il était parti dans une colère noire pour un simple quiproquo et elle n'avait pas pu le raisonner depuis. Même maintenant, la réconciliation semblait difficile : Bob venait d'ignorer purement et simplement son invitation à aller boire un café. Elle tenta une nouvelle stratégie et envoya « tu n'as pas répondu. » suivi quelques secondes plus tard d'un humble « tu m'en veux ? ».

L'histoire ne dit pas ce qui se passa ensuite, et nous ne pouvons que l'imaginer. Aussi étrange que cela puisse paraître, le destin d'Alice et de Bob dépend énormément du service de messagerie instantanée choisi pour communiquer. Nous avons

tenté de reproduire leur conversation avec trois services de messagerie instantanée grand-public : Google Hangouts[1] (Hangouts), WhatsApp Messenger[2] (WhatsApp) et Microsoft Skype[3] (Skype). L'expérience a été réalisée avec Matoula Petrolia dans le rôle d'Alice et moi-même dans le rôle de Bob. La perte de connexion de Bob a été modélisée par l'utilisation du mode avion des téléphones. Pour chaque expérience, nous présentons une capture d'écran prise avant reconnexion et une prise après reconnexion pour les deux interlocuteurs. Les trois services de messagerie présentent un comportement très différent face à la même situation.

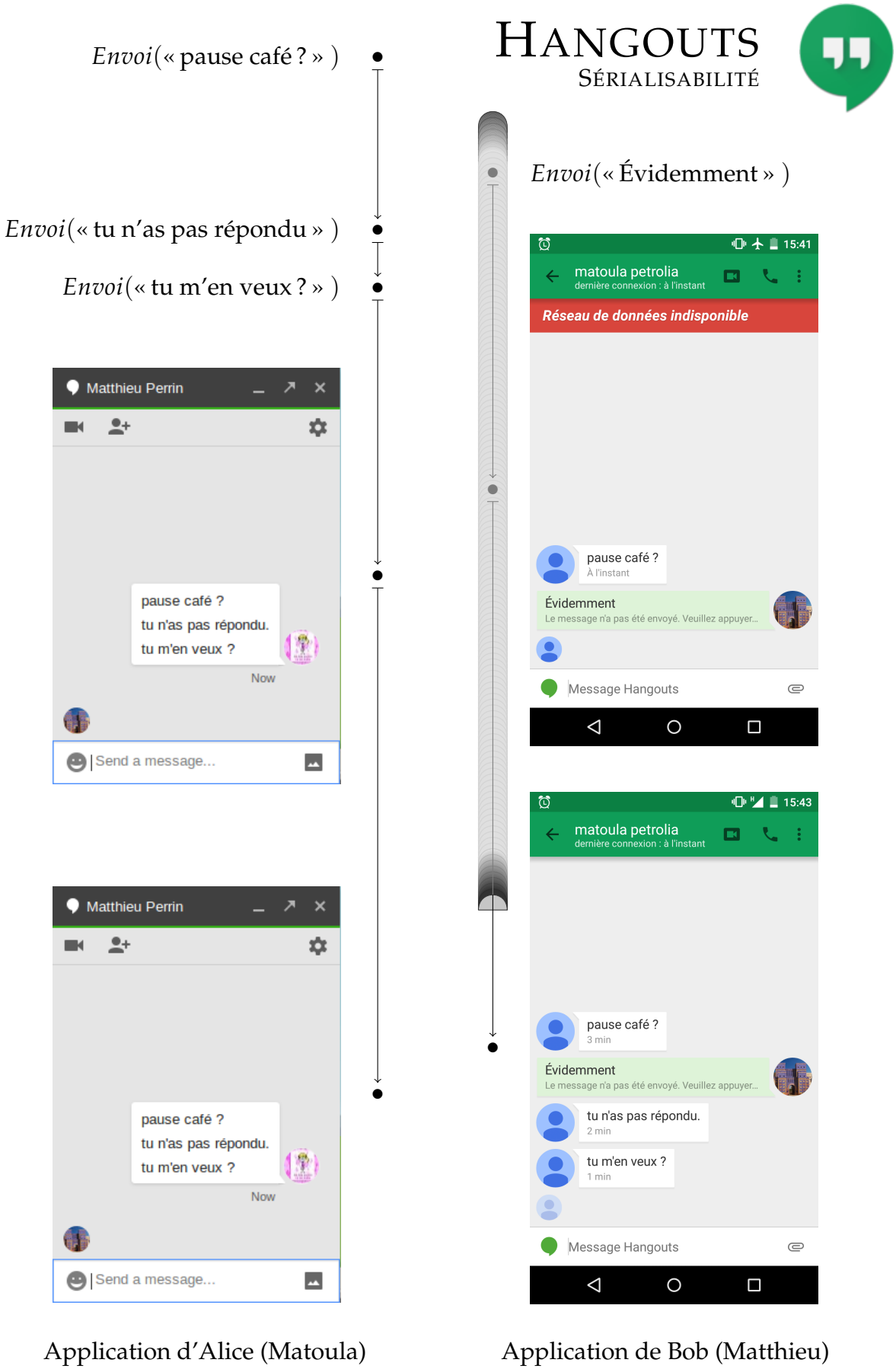
Hangouts (figure 1.1). Bob reçut les messages d'Alice en sortant du métro. Il les regarda distraitement en se disant que son propre message avait dû mettre du temps à lui parvenir. Ce n'est qu'en fin d'après-midi, alors qu'il s'apprêtait à préciser l'heure et le lieu du rendez-vous, qu'il remarque le message d'erreur écrit en petites lettres en dessous de sa réponse : « Le message n'a pas été envoyé. Veuillez appuyer pour réessayer. ». Il se résigna à appeler Alice pour mettre les choses au clair.

WhatsApp (figure 1.2). La réponse de Bob fit à Alice l'effet d'une douche froide. « Évidemment » ! Non seulement il lui en voulait, mais il ne se gênait pas pour le dire le plus sèchement possible. Bob lui demanda plus tard à quelle heure elle voulait le voir. Elle ne comprit pas tout de suite mais sauta sur l'occasion pour accepter. Ce n'est que le soir, quand Bob lui montra son propre fil de messages qu'elle comprit ce qui s'était passé : leurs messages s'étaient croisés et chacun avait reçu le message de l'autre après avoir envoyé le sien. Bob ne pouvait donc pas savoir qu'elle avait mal interprété son message.

Skype. (figure 1.3) La deuxième stratégie d'Alice fut aussi infructueuse que la première. Elle eut beau regarder régulièrement son téléphone toute la journée, aucun message de Bob ne vint se placer après les siens sur son fil de messages. Elle se sentit tout bête quand elle réalisa enfin son erreur : elle n'avait pas vu la réponse favorable que Bob lui avait envoyée avant même qu'elle ne lui envoie son deuxième message. Elle s'empressa de se confondre en excuses et de mettre au point une rencontre avec Bob.

À la lumière de ces trois scénarios, on voit que des compromis sont inévitables en cas de déconnexion temporaire, que ce soit un message d'erreur (Hangouts), un réordonnancement de messages (Skype) ou la présentation d'un état différent aux interlocuteurs (WhatsApp). Chaque stratégie a ses propres qualités mais aucune n'est exempte de défauts. Les services de messagerie instantanée ont vocation à être utilisées par des êtres humains, capables de s'adapter à beaucoup de situations. Les programmes, eux, le sont beaucoup moins et les conséquences d'incohérences imprévues dans des données partagées par des ordinateurs peuvent être beaucoup plus graves. Avoir un moyen efficace de modéliser précisément les différents types d'incohérence qui peuvent survenir est donc un enjeux très important.

[1] http://www.google.fr/hangouts/
[2] http://www.whatsapp.com/
[3] http://www.skype.com/fr/



Application d'Alice (Matoula) Application de Bob (Matthieu)

FIGURE 1.1 – Comportement de Google Hangouts après une déconnexion.

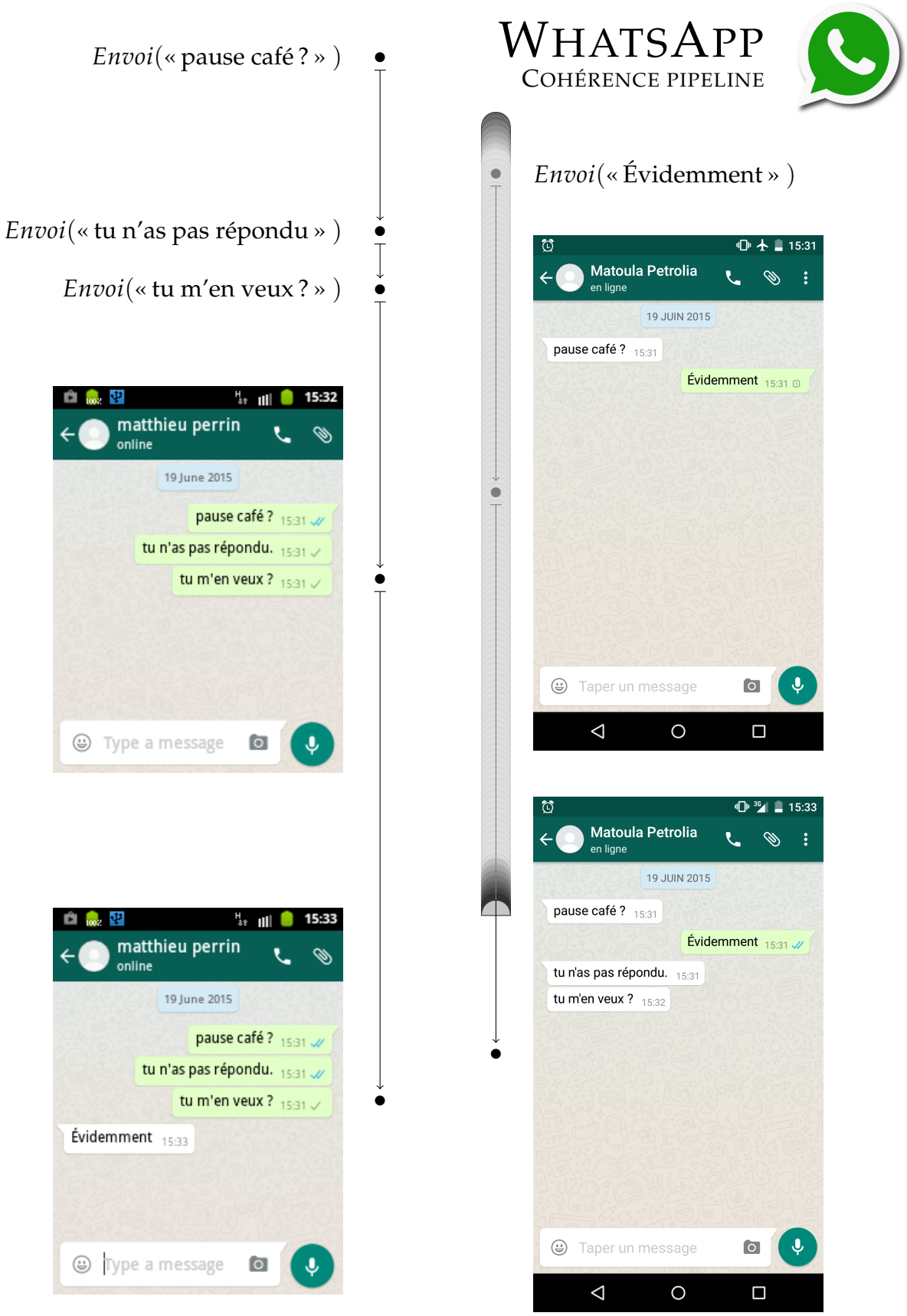
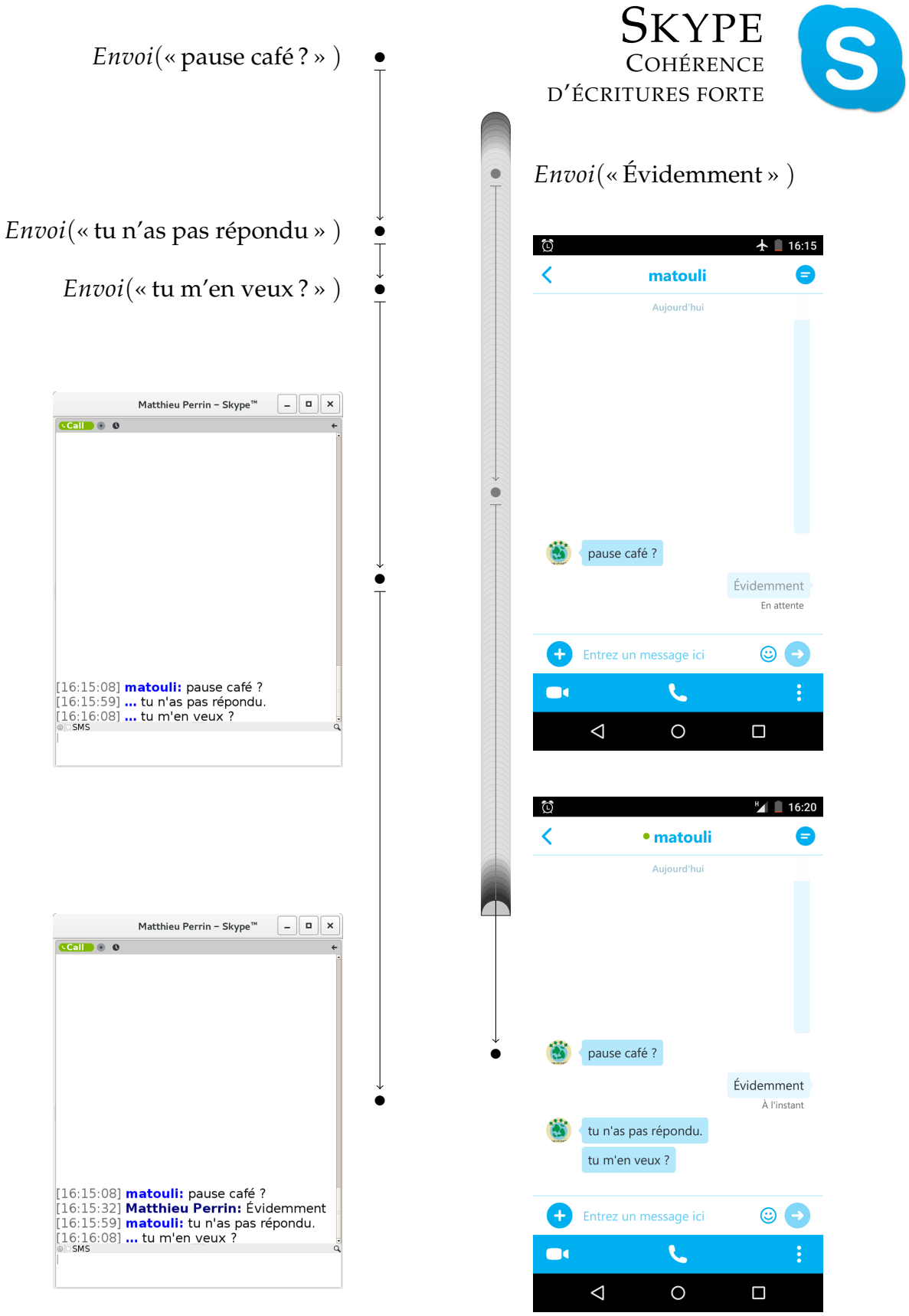


FIGURE 1.2 – Comportement de WhatsApp Messenger après une déconnexion.



Application d'Alice (Matoula) Application de Bob (Matthieu)

FIGURE 1.3 – Comportement de Microsoft Skype après une déconnexion.

1.1 Objets partagés

Un système réparti est une collection d'entités de calcul autonomes qui sont connectés en vue d'accomplir une tâche commune. Les systèmes répartis sont de plus en plus présents autour de nous. En 2015, le monde a dépassé les trois milliards d'internautes[4], le moindre processeur de téléphone portable est aujourd'hui multi-cœur, l'informatique décentralisée *en nuage* (« cloud computing ») commence à se démocratiser avec l'explosion des services qu'elle offre, la géo-réplication est largement utilisée pour garantir la persistance des données sensibles... Pourtant, force est de constater que la programmation d'applications à très grande échelle les exploitant reste une affaire de spécialistes. Certes le nombre de sites Web dans le monde se rapproche du milliard en 2015[5], mais l'architecture de chaque site est en fait très centralisée en raison du rôle prépondérant d'un serveur généralement séquentiel. Certes, des programmes phares comme SETI@home [116] ont vu le jour sur des architectures complètement décentralisées que sont les réseaux pair-à-pair, mais deux années d'effort ont été nécessaires à des spécialistes pour développer BOINC, la plateforme de gestion du projet. Certes, les langages de programmation modernes intègrent des mécanismes pour simplifier la gestion de la concurrence entre les fils d'exécution des programmes, mais la diversité des systèmes répartis empêche d'appliquer les mêmes outils à tous les systèmes.

C'est justement cette diversité qui rend la programmation répartie à grande échelle particulièrement complexe. Les raisons d'être des systèmes répartis sont nombreuses. On parle de *parallélisme* si le but est d'agréger de la puissance de calcul pour accomplir une tâche particulièrement complexe (simulation de phénomènes physiques ou biologiques...). Il peut également s'agir de *collaboration* (édition de documents, orchestration de sites Web...) ou au contraire de *compétition* (jeux en réseau, transactions à haute fréquence sur les marchés financiers...), voire de simple *communication* (services de messagerie, sites Web, partage de fichier en pair-à-pair...) entre entités indépendantes, généralement humaines. Enfin, la *réplication* peut être utilisée pour résister aux défaillances : si une machine tombe en panne, d'autres sont disponibles pour prendre le relais.

Malgré cette très grande variété, à très haut niveau, les applications réparties ont toutes un modèle semblable : des entités séquentielles, que nous nommerons *processus* à partir de maintenant, interagissent entre elles par l'intermédiaire d'un ou plusieurs *objets partagés* qui encapsulent les données partagées et gèrent la concurrence. Dans le cas d'un jeu en ligne, l'objet partagé est la partie en cours et les règles du jeu explicitent les opérations autorisées ; pour l'édition collaborative, il s'agit du document en cours d'édition ; les processus qui calculent les prévisions météorologiques partagent la carte en cours de simulation, etc. La mémoire partagée et les canaux de communication sont d'autres exemples d'objets partagés, mais à plus bas niveau. De manière générale, les applications réparties peuvent souvent être modélisées par couches, chaque couche étant une abstraction, sous forme d'objets partagés, des objets partagés de la couche précédente. Pour résumer, un programme peut être vu comme une hiérarchie d'objets partagé, chacun avec son niveau d'exigence sémantique. Les objets partagés ont donc une place centrale dans tous les domaines mettant en œuvre des systèmes répartis.

Le choix des algorithmes à mettre en œuvre dépend énormément de la qualité de service voulue et du système considéré, d'où l'intérêt de pouvoir évaluer précisément les comportements acceptés par une application et de les expliciter au sein d'une spécification précise. Les mêmes questions se posent à toutes les couches d'une application,

[4] http://www.internetlivestats.com/watch/internet-users/
[5] http://www.internetlivestats.com/total-number-of-websites/

du niveau de la mémoire partagée [23, 109] jusqu'au niveau applicatif, comme illustré dans le préambule.

Cette thèse se concentre plus précisément sur les *objets persistants*, ou « long-lived objects », qui sont le pendant partagé des instances de structures de données dans les systèmes séquentiels : ils encapsulent des données partagées et des opérations permettant de les modifier et de les lire, que les processus peuvent appeler quand ils le désirent. Nous laissons à des travaux futurs l'étude des *objets de décision à usage unique* (« one-shot objects », aussi appelés *tâches*) que chaque processus est censé invoquer exactement une fois, comme le Consensus ou le vote.

1.2 Systèmes répartis sans-attente

Un autre type de diversité doit être prise en compte quand on parle d'applications réparties : celle des systèmes sur lesquelles elles sont déployées. La programmation d'une simulation scientifique sur une grille de machines virtuelles ne met en effet pas du tout en jeu les mêmes mécanismes que celle d'un jeu massivement multijoueur en ligne qui s'appuie sur un serveur central, par exemple. Les systèmes répartis sont caractérisés par plusieurs paramètres.

Échelle du système. Un système est réparti à partir du moment où il contient au moins deux processus exécutés en parallèle. Un programme dont l'interface graphique s'affiche à l'écran de manière asynchrone peut donc déjà être considéré comme réparti. À l'opposé, Tianhe-2, actuellement le plus puissant superordinateur au monde, possède trois millions cent vingt mille cœurs et BOINC affirme être installé sur plus de treize millions d'ordinateurs dans le monde[6].

Moyen d'interaction. On distingue généralement les systèmes dans lesquels les processus doivent communiquer en s'échangeant des messages et ceux dans lesquels ils ont accès à une mémoire partagée. Des divisions plus fines peuvent détailler ces familles. Si les processus communiquent par messages, le réseau est-il complet ? Les processus connaissent-ils leurs voisins ? Ceux-ci sont-ils fixes ? Peuvent-ils les choisir ? Pour ceux communiquant par mémoire partagée, à quelles opérations ont-ils droit ? Peuvent-ils écrire dans tous les registres ?

Gestion des fautes. Plus les systèmes sont grands, plus les fautes sont fréquentes. Dans les systèmes pair-à-pair, des processus partent et d'autres se connectent en permanence ; dans un Internet de plus en plus composé de terminaux mobiles connectés, la fiabilité des transmissions ne peut pas être assurée ; les hackers chercheront toujours à attaquer les applications sensibles comme celles des institutions bancaires et gouvernementales (ce que l'on appelle fautes byzantines)... Ces fautes peuvent être de diverses natures : pannes franches (un processus arrête son exécution sans avertir les autres), pertes de messages (en émission ou en réception) ou comportements non conformes des processus (corruption de la mémoire ou attaques intentionnelles). Le nombre de fautes que l'on peut accepter et la présence d'un serveur fiable connu de tous sont également des caractéristiques déterminantes du système.

[6] http://boincstats.com/fr

Rapport au temps. La présence ou l'absence d'une horloge globale accessible par tous les processus change complètement la nature d'un système. Dans un système complètement synchrone, tous les processus avancent suivant un schéma temporel bien défini (éventuellement *a priori*) et, s'ils communiquent par messages, ces messages sont acheminés en un temps borné. Ces deux hypothèses peuvent être affaiblies. Dans un système complètement asynchrone, il n'y a pas de borne sur la vitesse relative des processus ni sur les délais de transfert des messages (*latence* du réseau). Si des pannes franches peuvent en plus se produire, un processus qui ne reçoit pas un message attendu est dans l'incapacité d'en déduire si l'émetteur est fautif ou extrêmement lent. L'hétérogénéité du matériel exécutant les processus peut être une cause importante d'asynchronisme.

Dans cette thèse, nous nous intéressons plus spécialement aux systèmes répartis dans lesquels les processus communiquent en s'envoyant des messages, mais dans lesquels il n'est pas acceptable ou possible pour un processus d'attendre explicitement des messages d'un autre processus. Cette hypothèse est réaliste pour modéliser de nombreux systèmes, dont ceux cités ci-dessous.

- Les systèmes dont le nombre de processus est inconnu, ce qui empêche de savoir le nombre de réponses à attendre, par exemple les systèmes pair-à-pair.
- Les systèmes pour lesquels le temps passé à attendre est jugé trop coûteux, par exemple pour le calcul à haute performance.
- Les systèmes dans lesquels des partitions peuvent se produire, empêchant momentanément les processus de communiquer entre eux. De telles partitions sont fréquentes dans les architectures en nuage [122], justifiant la pertinence du modèle sans-attente.
- Les systèmes pour lesquels la gestion des fautes est critique au point où le système doit continuer à fonctionner même si un processus se retrouve seul dans le système. Dans une telle situation, un processus ne peut pas attendre la participation d'un nombre connu *a priori* d'autres processus car ils peuvent être fautifs. C'est ce cas que nous privilégions pour modéliser l'hypothèse de l'absence d'attente car elle est plus simple à définir formellement.

Plus précisément, nous considérons les systèmes répartis asynchrones à passage de messages sans-attente. Nous ne donnons que l'intuition ici, ces systèmes étant définis formellement dans la section 2.1.2.

Systèmes répartis à passage de messages. Les systèmes que nous considérons sont composés d'un ensemble de taille fixe et connue de processus séquentiels qui communiquent grâce à des primitives d'envoi et de réception de messages.

Systèmes sans-attente. Tous les processus sont susceptibles de tomber en panne, et le système doit continuer à s'exécuter même réduit à un seul processus.

Systèmes asynchrones. Les processus ne s'exécutent pas tous à la même vitesse, et il n'y a pas de borne sur le temps mis par un message pour parvenir à sa destination. Dans ces conditions, un processus qui ne reçoit pas un message qu'il attend ne peut jamais savoir si ce message est très lent à lui parvenir ou si le message n'a jamais été envoyé en raison de la panne de l'émetteur.

Cette impossibilité d'attendre a des conséquences importantes sur les objets partagés que l'on peut y construire [12, 14]. Quand une opération sur un objet partagé est invoquée localement par un processus, elle doit être traitée en utilisant uniquement la connaissance locale du processus. Les lectures doivent être faites localement, ce qui implique que chaque processus ait une copie locale de l'objet. L'impossibilité de se synchroniser pendant les opérations empêche de supprimer toutes les incohérences. Celles-ci doivent donc être spécifiées avec une attention particulière.

1.3 Problématique

La question centrale qui guide cette thèse peut être formulée de cette façon :

Problématique. *Comment spécifier les objets partagés d'un système sans-attente ?*

Cette problématique se décompose en deux sous-problèmes : celui de la *spécification* et celui des *systèmes sans-attente*.

Qu'est-ce qu'une bonne spécification ? Lorsqu'un chef de projet conçoit l'architecture générale d'une application, l'un de ses principaux soucis est de séparer le travail en entités logiques distinctes, des briques logicielles, qui pourront être implémentées par des développeurs différents. Chaque brique logicielle est alors spécifiée séparément pour limiter les risques de conflit lors de l'intégration finale. La spécification est donc le vecteur de communication entre deux développeurs : le *concepteur* qui conçoit l'objet et l'*utilisateur* qui l'utilise dans son propre programme. Quand l'objet en question fait partie d'une bibliothèque (et d'autant plus quand le code source est fermé), c'est même la seule façon de décrire l'objet. Une bonne spécification doit être écrite du point de vue de l'utilisateur et pour l'utilisateur. Pour le concepteur, il s'agit plus d'un objectif à atteindre que d'une façon de décrire ce qu'il a réussi à faire (dans l'autre cas, on parle plus de *modélisation*). Nous décrivons maintenant les trois principales qualités que doit avoir une spécification.

Rigueur. Une spécification ne doit pas forcément être une caractérisation précise de toutes les réactions possibles de l'objet. Il s'agit en général plutôt de propriétés vérifiées dans toutes les situations. Le bon niveau de détails doit être adopté : des propriétés trop fortes contraignent trop l'implémentation, mais des contraintes trop faibles ne dépeignent que partiellement la sémantique de l'objet. Par contre, toutes les propriétés doivent être parfaitement définies mathématiquement pour ne pas laisser place à l'interprétation subjective. Cela est particulièrement vrai si des techniques de validation formelle sont utilisées.

Abstraction. Une façon de décrire le comportement d'un objet de manière rigoureuse est de donner son code source. Cela ne fait pourtant pas une bonne spécification. En effet, des modifications de code peuvent être nécessaires dans un but d'amélioration de l'efficacité, de résolution d'erreurs, ou d'adaptation à un nouvel environnement. Dans tous ces cas, si la sémantique reste la même, il n'y a aucune raison de changer la spécification. Une spécification doit être indépendante de l'implémentation. Pour un objet partagé, la valeur des données stockées sur la mémoire locale d'un processus et les messages envoyés, entre autres, ne doivent pas faire partie de la spécification.

Simplicité. Une spécification donne une abstraction de haut niveau à l'utilisateur de la brique logicielle. Ce que souhaite l'utilisateur qui utilise une structure de données de type pile est que la valeur qu'il dépile soit toujours la dernière valeur empilée qui n'ait pas été encore dépilée. Le fait que l'implémentation utilise des tableaux ou des listes chaînées lui est indifférent. Un utilisateur humain doit donc être en mesure de comprendre le plus rapidement possible à quoi sert la brique logicielle seulement en lisant sa spécification, d'où l'importance que la spécification soit conçue du point de vue du développeur qui utilise l'objet.

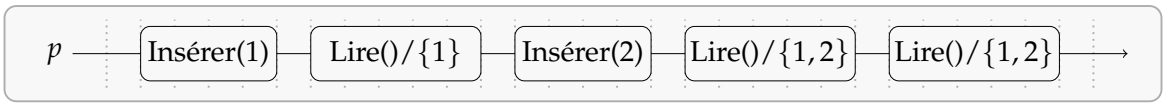
Quelles contraintes dans les systèmes sans-attente ? L'autre question concerne les systèmes que l'on considère plus spécifiquement. Quels objets partagés peut-on implémenter dans les systèmes sans-attente ? Quelles sont les propriétés que l'on peut en attendre ? Quelles sont les plus fortes que l'on peut imposer ?

1.4 Approche

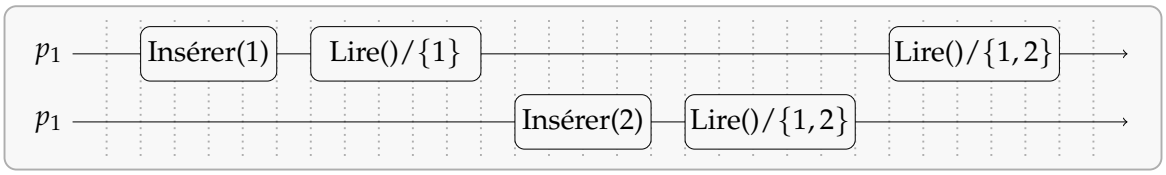
Nous avons vu que la spécification d'un objet partagé doit adopter le point de vue de l'utilisateur de l'objet partagé. Celui-ci veut en général utiliser une version partagée des objets dont il a l'habitude dans les programmes séquentiels. La spécification d'un objet partagé doit donc s'appuyer sur sa *spécification séquentielle*. Mais que signifie *la version partagée* ? Une exécution d'un programme qui n'est pas séquentielle est dite *concurrente*. Pendant une exécution concurrente, les accès aux objets partagés créent des interactions entre les exécutions locales des différents processus. La figure 1.4 montre quelques exemples d'histoires tirées d'exécutions concurrentes mettant en jeu un ou deux processus partageant un ensemble dans lequel il est possible d'insérer des valeurs, ou dont on peut lire le contenu, c'est-à-dire l'ensemble des valeurs précédemment insérées. La figure montre les opérations exécutées par chaque processus au cours du temps, qui s'écoule de la gauche vers la droite. Un rectangle arrondi délimite le temps entre l'appel et le retour d'une opération. Les valeurs retournées par les lectures sont représentées après une barre oblique.

Quelles histoires sont correctes ? Dans le cas séquentiel, il n'y a pas d'ambiguïté possible. Sur la figure 1.4a, lors de sa première lecture, le processus a inséré 1, il ne peut donc retourner que $\{1\}$. Les lectures suivant l'insertion du 2 n'ont d'autre choix que de retourner $\{1, 2\}$. La figure 1.4b doit également être considérée correcte : toutes les opérations sont consécutives dans le temps et la suite des valeurs retournées est la même que dans l'exécution séquentielle. Qu'en est-il de la figure 1.4c ? Les opérations sont également successives, mais la première lecture suit l'insertion du 2 sans la prendre en compte. D'un autre côté, chaque processus effectue exactement les mêmes opérations et lit les mêmes valeurs que dans l'exécution précédente. Or dans un système asynchrone, les processus ne peuvent pas faire la différence entre les deux histoires, car l'ordre temporel observé pourrait aussi bien être dû à une désynchronisation des horloges des deux processus. Il n'y a donc pas de réponse définitive à ce qu'est une histoire cohérente. À la place, on peut définir différents *critères de cohérence* qui sont chacun mieux adapté à certaines situations. L'histoire de la figure 1.4b est dite *linéarisable* alors que celle de la figure 1.4c est seulement *séquentiellement cohérente*.

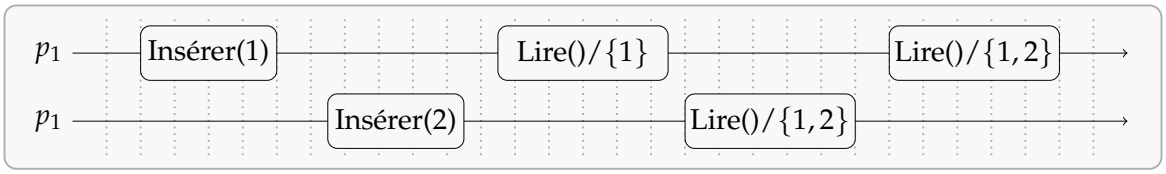
Hélas, la cohérence séquentielle et la linéarisabilité sont des critères trop forts pour être implémentés dans les systèmes sans-attente [14, 48] : des incohérences comme celle de l'histoire de la figure 1.4d ne peuvent pas être évitées. Or si la succession des



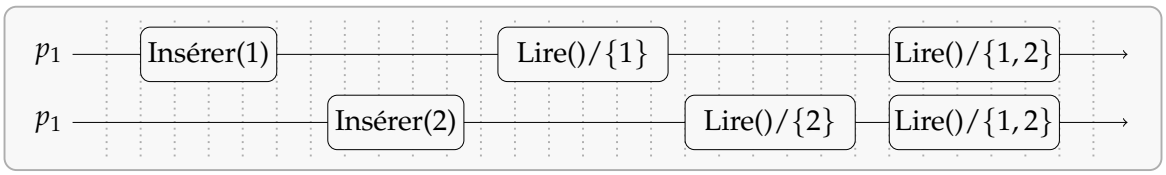
(a) Exécution séquentielle



(b) Exécution linéarisable



(c) Exécution séquentiellement cohérente



(d) Exécution causalement convergente

FIGURE 1.4 – Quels comportements sont acceptables ?

opérations n'a pas de sens, comment peut-on encore parler de spécification séquentielle ? Il est d'usage de penser que de tels objets ne peuvent pas être spécifiés sous la forme d'une spécification séquentielle et d'un critère de cohérence. Les travaux existants contournent le problème de deux manières. Soit ils restreignent leur analyse à la spécification de la mémoire partagée [6, 76], un type particulier composé de registres dans lesquels on ne peut qu'écrire et lire la dernière valeur écrite. Soit ils remplacent la spécification séquentielle par une spécification concurrente plus complexe qui mélange les aspects fonctionnels et ceux uniquement liés à la concurrence [28, 117]. Dans ce manuscrit, nous défendons la thèse inverse :

Thèse. *La spécification des objets partagés à l'aide de deux facettes complémentaires, une* spécification séquentielle *qui décrit la sémantique de l'objet et un* critère de cohérence *qui décrit comment la concurrence affecte son comportement, est pertinente pour les objets implémentables dans les systèmes sans-attente.*

S'ancrer dans l'existant est une priorité : la littérature sur le sujet présente un canevas serré de concepts et de définitions, souvent partiels et incomparables entre eux. L'extension et la formalisation de ces concepts est une partie importante du travail présenté ici. Nous nous efforçons de combler les vides pour extraire la structure de l'espace des critères faibles et d'en dresser une carte.

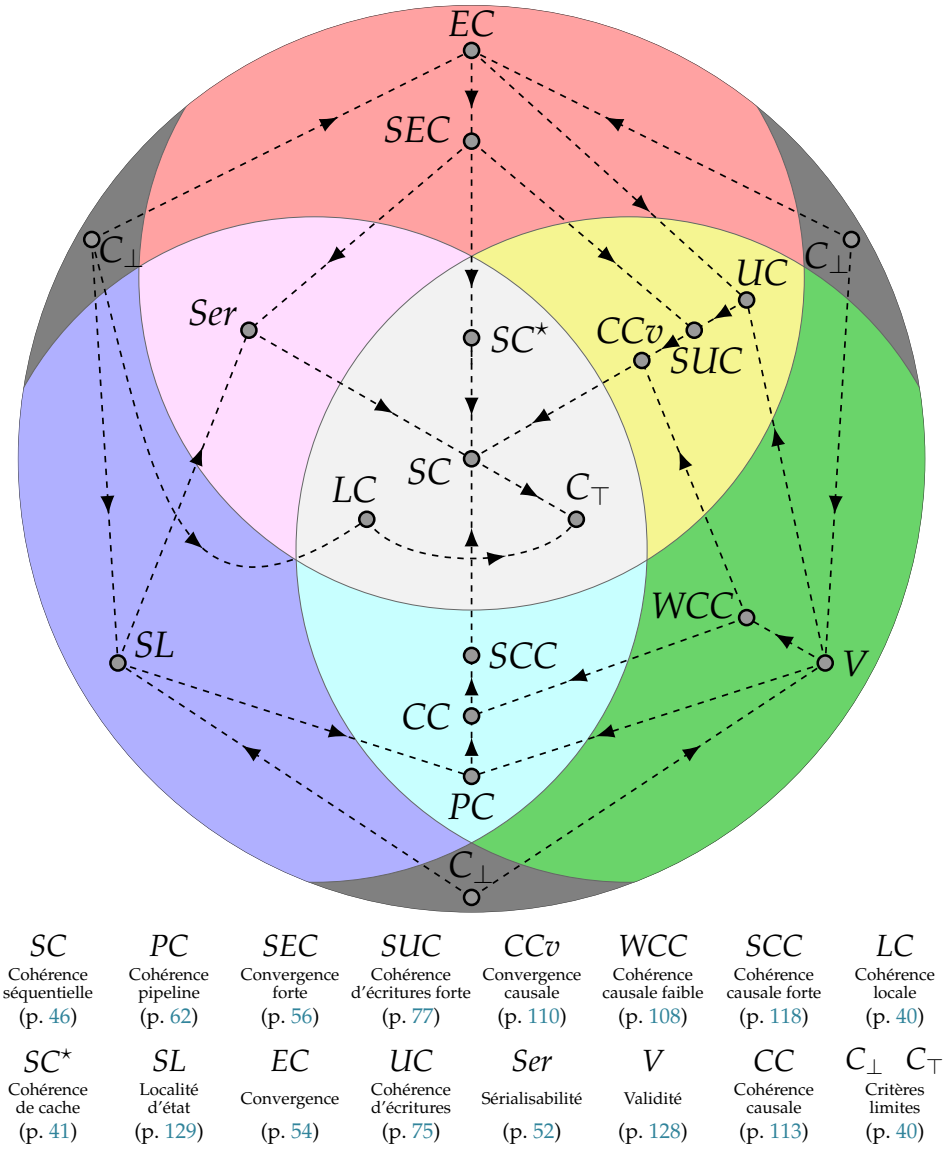


FIGURE 1.5 – Carte des critères de cohérence étudiés dans cette thèse.

1.5 Organisation et contributions

Après cette introduction, le chapitre 2 présente le cadre mathématique de notre travail : nous y définissons formellement les spécifications séquentielles, les histoires concurrentes, les critères de cohérence et les systèmes sans-attente $AS_n[\emptyset]$. Nous y étudions ensuite comment l'existant peut s'inscrire dans ce modèle et quelles difficultés doivent faire l'objet d'une étude plus poussée.

Le premier problème que nous rencontrons est la spécification de l'état de convergence des objets convergents [122]. Dans le chapitre 3, nous proposons deux nouveaux critères de cohérence : la *cohérence d'écritures* et la *cohérence d'écritures forte* qui relient l'état de convergence aux écritures grâce à la spécification séquentielle. Nous y présentons aussi trois nouveaux algorithmes pour les implémenter dans les systèmes sans-attente.

La mémoire causale [6] est reconnue comme un objet central parmi les modèles de mémoire qui peuvent être implémentés dans les systèmes sans-attente, mais sa définition utilise les liens sémantiques entre les lectures et les écritures. Faire entrer la

causalité comme un critère de cohérence dans notre modèle est un défi relevé par le chapitre 4. Nous y définissons quatre critères de cohérence. La *cohérence causale faible* est le plus grand dénominateur commun de ces quatre critères. Elle peut être associée à la cohérence d'écritures pour former la *convergence causale* ou à la cohérence pipeline pour former la *cohérence causale*, qui correspond à la mémoire causale quand elle est appliquée à la mémoire. La *cohérence causale forte* enfin permet l'étude de la fausse causalité induite par l'implémentation standard de la mémoire causale.

Le chapitre 5 se concentre sur la calculabilité dans les systèmes sans-attente. Nous y dressons une cartographie de la structure de l'espace des critères faibles, divisée en trois familles de critères primaires (*convergence*, *validité* et *localité d'état*) qui peuvent être conjugués deux à deux pour former trois familles de critères secondaires (*cohérence d'écritures*, *cohérence pipeline* et *sérialisabilité*).

Le chapitre 6 présente la bibliothèque *CODS* qui met en œuvre les critères de cohérence faibles dans le langage de programmation orienté objet D. CODS offre une interface de programmation abstraite la plus transparente possible, dans laquelle l'instanciation est seulement remplacée par l'évocation d'une spécification séquentielle définie par une classe du programme et d'un critère de cohérence parmi ceux de la bibliothèque ou bien implémenté comme une extension de cette bibliothèque.

Les techniques de spécification introduites sont indépendantes du système dans lequel l'objet est implémenté. Ce système est seulement modélisé par des *histoires concurrentes*. Dans le chapitre 7, nous montrons l'universalité de cette modélisation en l'appliquant aux orchestrations de services Web définies à l'aide du langage Orc [66]. Pour cela, nous définissons une nouvelle sémantique opérationnelle instrumentée qui calcule les dépendances entre les événements pendant l'exécution, de sorte à construire ces histoires concurrentes.

Contributions.

1. *Nous proposons un nouveau formalisme pour spécifier les objets partagés sous la forme d'une spécification séquentielle et d'un critère de cohérence.*
2. *Nous montrons que ce formalisme s'intègre élégamment dans les langages de programmation orientés objets en présentant la bibliothèque CODS écrite pour le langage D.*
3. *Nous réécrivons plusieurs critères de cohérence existants, dont la cohérence causale, dans ce formalisme, ce qui impose d'étendre leur définition à toutes les spécifications séquentielles.*
4. *Nous définissons plusieurs nouveaux critères de cohérence : plusieurs variantes de la cohérence d'écritures et de la cohérence causale, la validité et la localité d'état.*
5. *Nous dressons une cartographie plus précise de l'espace des critères faibles dans laquelle s'inscrivent tous les critères que nous définissons ou généralisons, illustrée par la figure 1.5.*
6. *Nous proposons une nouvelle classe de structures de données, les flux fenêtrés, qui remplissent tous les niveaux de la hiérarchie de Herlihy [56].*
7. *Nous mettons au point une sémantique opérationnelle instrumentée pour le langage Orc, qui calcule les dépendances entre événements pendant l'exécution.*

Le chapitre 8 conclut ce manuscrit et expose les pistes de recherche futures. Après la bibliographie, les tables des figures et des notations aident à se retrouver dans le document.

2

Spécifier les objets partagés : analyse comparée de l'état de l'art

Sommaire

Introduction

Dans les systèmes répartis, les objets partagés sont les outils qui permettent aux processus de communiquer entre eux. En dehors du contexte séquentiel, les appels aux opérations sur ces objets ne sont pas totalement ordonnés. On parle alors de concurrence. Le problème de la spécification des objets partagés est central dans tous les domaines qui mettent en jeu de la concurrence, tels l'algorithmique du réparti, les systèmes d'exploitation, la programmation parallèle ou encore les systèmes de gestion de bases de données ou de mémoires transactionnelles.

Dans [113], Shavit a clairement résumé la façon dont les objets partagés devraient être spécifiés selon lui. « Il est infiniment plus simple et plus intuitif pour nous humains de spécifier comment des structures de données abstraites se comportent dans

un cadre séquentiel, où il n'y a pas d'entrelacement. Ainsi, l'approche standard pour exprimer les propriétés de sûreté d'une structure de données concurrente est de spécifier les propriétés de la structure séquentiellement, et de trouver une façon de lier les exécutions concurrentes à ces exécutions séquentielles "correctes". »

Cette vision de la spécification des objets partagée est répandue au sein de la communauté de l'algorithmique du réparti. Le plus souvent, les objets ciblés dans ce domaine cachent la concurrence en se comportant comme si tous les accès étaient séquentiels. On dit qu'ils vérifient la *cohérence forte*. Il est donc naturel de séparer la spécification séquentielle de tels objets en une *spécification séquentielle* et un *critère de cohérence*. Le problème est que ces objets ne peuvent pas être construits sans-attente [48], et donc la cohérence forte n'est souvent pas adaptée pour spécifier les objets partagés des systèmes sans-attente. Par exemple, dans les applications accessibles par plusieurs utilisateurs parfois géographiquement distants, comme les éditeurs collaboratifs de documents [85, 87, 97, 124] et les entrepôts de données [35, 70], les données sont répliquées sur tous les processus ou une partie d'entre eux. La cohérence forte n'est généralement pas vérifiée : à la place, ces applications garantissent que, si tous les participants cessent de modifier l'objet, tous les réplicas finiront par converger dans un état commun. Il est donc nécessaire de spécifier l'état dans lequel elles convergent. Similairement, Des modèles de mémoire faiblement cohérente ont été proposés [6, 76] pour la programmation parallèle, à la fois pour limiter le temps des accès en lecture et en écriture, et pour des raisons de coût de fabrication et de mise au point. Les algorithmes ne fonctionnant pas de la même manière sur les différents modèles de mémoire, il est nécessaire de spécifier précisément ces nouveaux modèles. Chacune des communautés citées ci-dessus a proposé ses propres approches et outils de spécification des objets partagés.

Problème. *Deux problèmes se posent au vu de l'ampleur des travaux réalisés sur la spécification des objets répartis.*

- *Quelles solutions ont été proposées par les communautés de chacun de ces domaines pour spécifier leurs objets partagés ?*
- *Comment comparer ces solutions ?*

Des analyses de l'existant ont été réalisés séparément pour chacun de ces domaines mais, à notre connaissance, aucun état de l'art de cette ampleur n'a cherché à unifier les travaux de toutes ces communautés. La difficulté vient du fait que les travaux des différentes communautés ont des buts différents, qui les ont poussées à développer des outils et des formalismes différents. Pour illustrer ceci, reprenons l'exemple du comportement observé dans l'introduction avec le service de messagerie instantanée WhatsApp (figure 1.2). La convergence n'est pas implémentée (les messages apparaissent dans des ordres différents chez les deux interlocuteurs) donc ni le formalisme de l'algorithmique du réparti ni celui des systèmes collaboratifs ne permet de le spécifier correctement ; les opérations (l'envoi d'un message et l'affichage des messages reçus) ne correspondent pas à la lecture et à l'écriture dans une mémoire ; enfin, l'application n'a pas été développée à base de transactions donc la théorie des systèmes transactionnels ne s'applique pas.

Approche. *Nous proposons de fixer un cadre commun suffisamment large pour exprimer les subtilités des notions proposées par les différentes communautés, puis d'étendre les définitions existantes à ce cadre.*

Ce cadre commun est inspiré de la façon dont sont spécifiés les objets partagés dans la communauté de l'algorithmique du réparti. Nous séparons la spécification des objets

partagés en deux facettes distinctes et complémentaires : une *spécification séquentielle* et un *critère de cohérence*. La spécification séquentielle décrit les aspects fonctionnels de l'objet sans tenir compte des contraintes liées à la concurrence. Dans le cas des services de messagerie instantanée (que ce soit Hangouts, WhatsApp ou Skype), la spécification séquentielle décrit simplement le fait que les messages sont affichés dans l'ordre où ils sont envoyés. Nous décrivons ici cette spécification à l'aide d'un langage formel reconnu par un système de transitions, le *type de données abstrait*. Le critère de cohérence est une fonction qui transforme les spécifications séquentielles en ensembles d'*histoires concurrentes* admises par l'objet partagé. Une histoire concurrente est la modélisation de l'exécution d'un programme sur un *système réparti* donné, qui décrit les entités de calcul et leur façon de communiquer. Elle se présente sous la forme d'un ensemble d'événements partiellement ordonnés par un *ordre de processus* qui décrit la séquentialité des entités de calcul du système et par un *ordre de programme* qui peut décrire des relations plus complexes, comme la préemption. Nous supposons dans ce chapitre, en dehors de la section 2.1.2, que la modélisation du système réparti à l'aide des histoires concurrentes a déjà été réalisé.

Ce cadre permet par exemple de spécifier les modèles de mémoire : la mémoire y est vue comme un type de données abstrait particulier et les modèles de mémoire seront chacun étendu en un critère de cohérence qui, appliqué au type mémoire, redonne le modèle de mémoire d'origine. Le travail d'extension des modèles de mémoire aux autres types de données abstraits est l'une des contributions de ce chapitre.

Contribution. *Les contributions présentées dans ce chapitre sont de deux types.*

- *Le cadre que nous proposons est, à notre connaissance, le premier modèle unifiant les critères de cohérence à cette échelle.*
- *Nous étendons plusieurs critères de cohérence, initialement seulement définis pour la mémoire, aux autres spécifications séquentielles : la mémoire PRAM, la cohérence de cache et la mémoire lente.*

Ce chapitre est organisé en deux grandes parties. La partie 2.1 présente notre cadre commun, en en particulier les notions de type de données abstrait, d'histoire concurrente et de critère de cohérence et la façon dont nous les modélisons. Ensuite, la partie 2.2 présente l'analyse de l'existant dans quatre domaines : l'algorithmique du réparti, la programmation parallèle, les systèmes répartis et les systèmes transactionnels. Pour chaque domaine, nous identifions les critères de cohérence les plus importants et les exprimons dans notre cadre.

2.1 Spécifier les objets partagés

2.1.1 Spécifications séquentielles

La spécification séquentielle d'un objet décrit le comportement de cet objet lorsqu'il est exécuté dans un environnement séquentiel (c'est-à-dire quand il n'est utilisé que par un seul processus). Les travaux qui s'intéressent à la spécification des types de données se ramènent, *in fine* à décrire des propriétés sur des automates et des langages formels. Dans cette thèse, nous utilisons aussi des systèmes de transitions pour spécifier les types de données abstraits. La spécification séquentielle d'un type de données abstrait est le langage qu'il reconnaît.

Nous illustrons cette partie par quatre types de données abstraits : l'ensemble (figure 2.1), le flux fenêtré (figure 2.2), la file (figure 2.3) et la mémoire (figure 2.4).

Types de données abstraits

Le modèle que nous utilisons est une forme de transducteur très proche des machines de Mealy [79], à la différence que l'on accepte un nombre infini — mais dénombrable — d'états. Les valeurs qui peuvent être prises par le type de données sont encodées dans l'*état abstrait*, pris dans un ensemble Z. Il est possible d'accéder à l'objet en utilisant les symboles d'un *alphabet d'entrée* A. À la différence des méthodes d'une classe, les symboles d'entrée du type de données abstrait ne possèdent pas d'arguments. En effet, comme on s'autorise un ensemble potentiellement infini d'opérations, l'appel de la même méthode avec des arguments différents est encodée par des symboles différents. Par exemple, l'ensemble (figure 2.1) possède autant de symboles d'entrée d'insertion $I(v)$ qu'il y a d'éléments v à insérer. Tous ces symboles correspondraient à la même méthode I appelée avec un argument dans une implémentation. Une méthode peut avoir deux types d'effets. D'un côté, elle peut avoir un effet de bord, qui change l'état abstrait. La transition correspondante dans le système de transitions est formalisée par une *fonction de transition* τ. D'un autre côté, les méthodes peuvent avoir une valeur de retour prise dans un *alphabet de sortie* B, qui dépend de l'état dans lequel elles sont appelées et d'une *fonction de sortie* δ. Par exemple, la méthode *pop* dans une pile enlève l'élément au sommet de la pile (c'est son effet de bord) et retourne cet élément (c'est sa sortie). La définition formelle des types de données abstraits est donnée dans la définition 2.1.

Définition 2.1 (Type de données abstrait)**.** Un *type de données abstrait* (ADT, pour *abstract data type*) est un sextuplet $T = (\mathsf{A}, \mathsf{B}, \mathsf{Z}, \zeta_0, \tau, \delta)$ où :

- A et B sont des ensembles dénombrables appelés *alphabet d'entrée* et *alphabet de sortie*.
- Z est un ensemble dénombrable d'*états abstraits* et $\zeta_0 \in \mathsf{Z}$ est l'*état initial* ;
- $\tau : \mathsf{Z} \times \mathsf{A} \to \mathsf{Z}$ est la *fonction de transition* ;
- $\delta : \mathsf{Z} \times \mathsf{A} \to \mathsf{B}$ est la *fonction de sortie*.

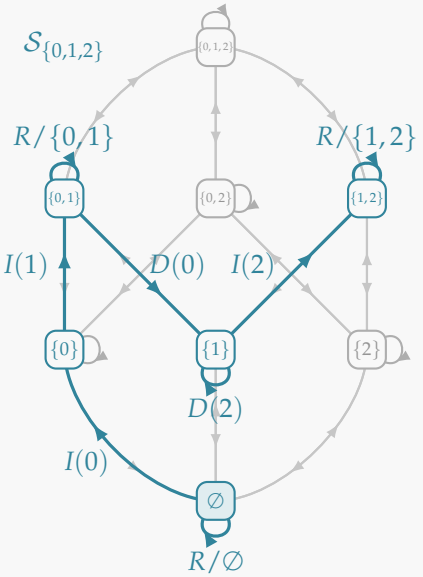
Nous considérons les types de données abstraits à isomorphisme près, car le nom des états n'est jamais utilisé directement. De plus, nous supposons que les fonctions τ et δ sont calculables par une machine de Turing pour s'assurer que les difficultés d'implémentation sont uniquement liées à la gestion de la concurrence. L'ensemble des ADT est noté $\mathcal{T}$.

La fonction de transition est complète, ce qui exprime le fait que l'objet réagit à toutes les sollicitations extérieures imposées par l'utilisateur. On peut modéliser le fait qu'un appel est incongru à un moment donné (par exemple lors d'une tentative de placer un pion blanc pendant le tour du joueur noir au Go) par une boucle sur certains états ou par un état d'erreur. Dans l'ADT ensemble, supprimer un objet qui n'a jamais été inséré n'a aucun effet. Cela est pris en compte par le fait que $\tau(\zeta, D(v)) = \zeta$ si $v \notin \zeta$. Le système de transitions est également déterministe, ce qui empêche de modéliser des objets comme certains types de tas qui ont une méthode pour retourner n'importe quel objet inséré. Il serait possible d'étendre le modèle aux ADT probabilistes ou non-déterministes, mais nous avons choisi de ne pas le faire ici pour ne pas apporter de complexité non directement liée au sujet abordé.

Voyons maintenant comment modéliser les services de messagerie instantanée des expériences de l'introduction sous la forme de types de données abstraits. Ils possèdent

ENSEMBLE

L'ensemble (définition 2.2) est une structure de données importante car elle est à la base de la plupart des systèmes de gestion de données. Il contient des éléments pris dans un ensemble dénombrable d'éléments Val appelé support. *Le type de données abstrait $\mathcal{S}_{Val}$ offre trois sortes d'opération : pour chaque élément v de Val, une opération d'insertion $I(v)$ et une opération de suppression $D(v)$, ainsi qu'une opération de lecture globale R qui retourne tous les éléments qui appartiennent à l'ensemble. Initialement, l'ensemble est vide et quand un élément est ajouté, il devient présent jusqu'à ce qu'il soit supprimé. La figure ci contre représente graphiquement le système de transitions correspondant à $\mathcal{S}_{\{0,1,2\}}$. Les valeurs de retour $\perp$ des opérations d'insertion et de suppression n'y sont pas représentées. Un chemin du système de transitions est représenté en bleu : le mot formé par la suite des étiquettes sur ce chemin fait partie de la spécification séquentielle de l'ensemble.*



Définition 2.2 (Ensemble)**.** Soit *Val* un ensemble dénombrable. L'ensemble de support *Val* est le type de données abstrait :

$$\mathcal{S}_{Val} = \left(\mathsf{A} = \bigcup_{v \in Val} \{\mathrm{I}(v); \mathrm{D}(v); \mathrm{R}\}, \mathsf{B} = \mathcal{P}_{<\infty}(Val)^{1} \cup \{\perp\}, \mathsf{Z} = \mathcal{P}_{<\infty}(Val), \zeta_0 = \emptyset, \tau, \delta\right)$$

$$\tau : \begin{cases} \mathsf{Z} \times \mathsf{A} & \to & \mathsf{Z} \\ (\zeta, \mathrm{I}(v)) & \mapsto & \zeta \cup \{v\} \\ (\zeta, \mathrm{D}(v)) & \mapsto & \zeta \setminus \{v\} \\ (\zeta, \mathrm{R}) & \mapsto & \zeta \end{cases} \qquad \delta : \begin{cases} \mathsf{Z} \times \mathsf{A} & \to & \mathsf{B} \\ (\zeta, \mathrm{I}(v)) & \mapsto & \perp \\ (\zeta, \mathrm{D}(v)) & \mapsto & \perp \\ (\zeta, \mathrm{R}) & \mapsto & \zeta \end{cases}$$

FIGURE 2.1 – L'ensemble.

une opération d'écriture pour l'envoi de chaque message et une opération de lecture qui affiche le résultat à l'écran. Si l'on ne modélise que le contenu de l'écran, seules les derniers messages sont retournés lors d'une lecture, affichés du plus ancien au plus récent. Nous proposons de modéliser les services de messagerie instantanée par des *flux fenêtrés*, formellement définis sur la figure 2.2. La taille k du flux fenêtré représente le nombre de messages qui peuvent être affichés sur l'écran. L'envoi d'un message, comme l'écriture dans le flux fenêtré, place l'argument à la fin de la file des messages et la lecture retourne les k dernières valeurs écrites.

Spécifications séquentielles

La succession des appels des méthodes d'un objet par un programme séquentiel forme une séquence d'opérations admise par l'objet. Un type de données abstrait définit la spécification séquentielle d'un objet, c'est-à-dire l'ensemble des histoires séquentielles admises par l'objet qu'il décrit. La façon naïve de décrire le langage reconnu par un ADT est de considérer l'ensemble des chemins infinis qui parcourent son système de transitions. Cet ensemble est trop restrictif dans le contexte de ce travail. Dans la *spéci-*

[1] $\mathcal{P}_{<\infty}(E)$ désigne l'ensemble des parties finies de E (voir table des notations).

fication séquentielle (définition 2.4), nous autorisons que la connaissance d'une histoire soit partielle. Aussi, nous acceptons les histoires finies, c'est-à-dire les préfixes d'histoires infinies, ainsi que les histoires dans lesquelles certaines valeurs de sortie sont inconnues. La spécification séquentielle est donc un langage de mots finis et infinis sur un alphabet d'*opérations* (définition 2.3) comportant les symboles d'entrée parfois associés à un symbole de sortie.

Définition 2.3 (Opération)**.** Soit $T = (\mathsf{A}, \mathsf{B}, \mathsf{Z}, \zeta_0, \tau, \delta)$ un type de données abstrait.

Une *opération* de T est un élément de $\Sigma = \mathsf{A} \cup (\mathsf{A} \times \mathsf{B})$. Un couple $(\alpha, \beta) \in \mathsf{A} \times \mathsf{B}$ est noté α/β.

Nous étendons la fonction de transition τ sur les opérations en appliquant τ sur le symbole d'entrée des opérations :

$$\tau_T : \begin{cases} \mathsf{Z} \times \Sigma & \to & \mathsf{Z} & \\ (\zeta, \alpha) & \mapsto & \tau(\zeta, \alpha) & \text{si } \alpha \in \mathsf{A} \\ (\zeta, \alpha/\beta) & \mapsto & \tau(\zeta, \alpha) & \text{si } \alpha/\beta \in \mathsf{A} \times \mathsf{B} \end{cases}$$

Nous définissons également la fonction δ_T^{-1} qui retourne l'ensemble des états de T dans lesquels une opération donnée peut être réalisée :

$$\delta_T^{-1} : \begin{cases} \Sigma & \to & \mathcal{P}(\mathsf{Z}) & \\ \alpha & \mapsto & \mathsf{Z} & \text{si } \alpha \in \mathsf{A} \\ \alpha/\beta & \mapsto & \{\zeta \in \mathsf{Z} : \delta(\zeta, \alpha) = \beta\} & \text{si } \alpha/\beta \in \mathsf{A} \times \mathsf{B} \end{cases}$$

Remarque 2.1. Un symbole de A peut être désigné, suivant le contexte, par les termes « opération », « méthode » et « symbole de l'alphabet d'entrée ». Lorsqu'il n'y a pas d'ambiguïté, nous privilégierons le terme « opération » (en particulier pour décrire les algorithmes), moins connoté relativement aux langages de programmation que le terme « méthode » et plus explicite en français que le terme « symbole de l'alphabet d'entrée ».

Définition 2.4 (Spécification séquentielle d'un ADT)**.** Une séquence finie ou infinie $\sigma = (\sigma_i)_{i \in D} \in \Sigma^\infty$, où D est soit $\mathbb{N}$ soit $\{0, ..., |\sigma| - 1\}$, est une *histoire séquentielle* d'un ADT T s'il existe une séquence de même longueur $(\zeta_{i+1})_{i \in D} \in \mathsf{Z}^\infty$ (ζ_0 étant déjà défini comme l'état initial) d'états de T tels que, pour tout $i \in D$,

- le symbole de sortie éventuel de σ_i est compatible avec ζ_i : $\zeta_i \in \delta_T^{-1}(\sigma_i)$
- l'exécution de l'opération σ_i fait passer de l'état ζ_i à l'état ζ_{i+1} : $\tau_T(\zeta_i, \sigma_i) = \zeta_{i+1}$

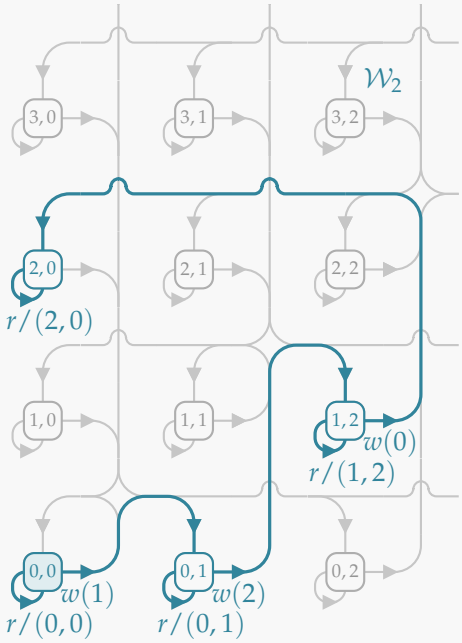
La *spécification séquentielle* de T est l'ensemble $L(T)$ de ses histoires séquentielles.

Lectures et écritures

Il est souvent utile de classer les opérations selon leurs effets. L'effet de bord d'une opération sera appelée sa partie *écriture* et le lien entre l'état et sa valeur de retour sera appelé sa partie *lecture*. Certaines opérations, qui n'ont qu'une partie lecture ou écriture, sont appelées *pures*. La définition 2.6 classe donc les opérations en quatre catégories : les lectures pures qui ne modifient pas l'état abstrait opposées aux écritures et les écritures pures pour lesquelles la valeur retournée est indépendante de l'état, opposées aux lectures.

FLUX FENÊTRÉ

Les flux fenêtrés occupent une place importante dans cette thèse car ils sont très souvent utilisés à titre d'exemple. Un flux fenêtré de taille k, $\mathcal{W}_k$ (définition 2.5), offre une opération d'écriture $w(n)$ pour tout $n \in \mathbb{N}$ et une opération de lecture r qui retourne la séquence des k dernières valeurs écrites si elles existent, les valeurs manquantes étant remplacées par la valeur par défaut 0.
La figure ci-contre montre un morceau du système de transitions d'un flux fenêtré de taille 2. Pour raison de clarté, les transitions sortant et rentrant dans un état ont été regroupées. Après l'écriture du 1, l'état $(0,1)$ est constitué de l'unique valeur 1 précédée de la valeur par défaut 0. Après l'écriture d'un 2, la première valeur est décalée pour faire place au 2 et le flux fenêtré passe dans l'état $(1,2)$. Si l'on écrit un 0, la plus ancienne valeur écrite, 1, sort alors de la fenêtre et l'état devient $(2,0)$.



Définition 2.5. Soit $k \in \mathbb{N}$. Le *flux fenêtré* entier de taille k est l'ADT :

$$\mathcal{W}_k = \Big(\mathsf{A} = \{r, w(n) : n \in \mathbb{N}\}, \mathsf{B} = \mathbb{N}^k \cup \{\bot\}, \mathsf{Z} = \mathbb{N}^k, \zeta_0 = (0,\dots,0), \tau, \delta\Big)$$

$$\tau : \begin{cases} \mathsf{Z} \times \mathsf{A} & \to & \mathsf{Z} \\ ((n_1,\dots,n_k), \mathrm{w}(x)) & \mapsto & (n_2,\dots,n_k,x) \\ ((n_1,\dots,n_k), \mathrm{r}) & \mapsto & (n_1,\dots,n_k) \end{cases} \qquad \delta : \begin{cases} \mathsf{Z} \times \mathsf{A} & \to & \mathsf{B} \\ ((n_1,\dots,n_k), \mathrm{w}(x)) & \mapsto & \bot \\ ((n_1,\dots,n_k), \mathrm{r}) & \mapsto & (n_1,\dots,n_k) \end{cases}$$

FIGURE 2.2 – Le flux fenêtré.

Définition 2.6 (Lectures et écritures)**.** Soient un ADT $T = (\mathsf{A}, \mathsf{B}, \mathsf{Z}, \zeta_0, \tau, \delta)$ et un symbole d'entrée $\alpha \in \mathsf{A}$.

- α est une *lecture pure* si elle n'a pas d'effet de bord, c'est-à-dire $\forall \zeta \in \mathsf{Z}, \tau(\zeta, \alpha) = \zeta$. L'ensemble des lectures pures de T est noté $\hat{Q}_T$ (pour *query*).
- α est une *écriture pure* si la valeur qu'elle retourne, dite *muette*, est indépendante de l'état, c'est-à-dire $\exists \beta \in \mathsf{B}, \forall \zeta \in \mathsf{Z}, \delta(\zeta, \alpha) = \beta$. L'ensemble des écritures pures de T est noté $\hat{U}_T$ (pour *update*). Quand il n'y a pas de risque d'ambiguïté entre α/β et α, nous omettrons le symbole de retour dans les écritures pures dans le but de simplifier les notations.
- α est une *lecture* si ce n'est pas une écriture pure, c'est-à-dire si $\alpha \in Q_T = \mathsf{A} \setminus \hat{U}_T$.
- α est une *écriture* si ce n'est pas une lecture pure, c'est-à-dire si $\alpha \in U_T = \mathsf{A} \setminus \hat{Q}_T$.

Par exemple, les opérations d'insertion et de suppression de l'ensemble sont des écritures pures et R est une lecture pure. Il existe également des opérations qui ne sont ni des lectures pures ni des écritures pures. C'est par exemple le cas de l'opération *pop* de la file (figure 2.3) qui possède à la fois une partie écriture car elle supprime la tête de file et une partie lecture car elle retourne cet élément. Les opérations qui sont à la fois des lectures pures et des écritures pures sont en général des opérations d'introspection

qui retournent des informations sur le type lui-même, comme la méthode `getClass` en Java. Ces opérations sont moins intéressantes à étudier car elles ne posent pas de problème lié à la concurrence.

Un *type de données abstrait à lectures et écritures* (UQ-ADT, pour *Update-Query Abstract Data Type*) T est un ADT particulier pour lequel chaque opération est soit une lecture pure soit une écriture pure, c'est-à-dire $\hat{U}_T \cup \hat{Q}_T = \mathsf{A}$ et $U_T \cap Q_T = \emptyset$. L'ensemble en est un exemple. L'ensemble des UQ-ADT est noté $\mathcal{T}_{UQ}$. Il est toujours possible de transformer un ADT quelconque en UQ-ADT en séparant la partie lecture de la partie écriture dans les opérations qui sont les deux à la fois. Pour reprendre l'exemple de la file, l'opération *pop* peut être scindée en une opération *hd* qui lit seulement la tête de file sans y toucher (lecture pure) et une opération *rh* qui supprime la tête de file sans en retourner la valeur (écriture pure). La définition 2.7 explicite une procédure de transformation générique. Les opérations α qui correspondent à la fois à une lecture et une écriture y sont scindées en une lecture pure α^Q et une écriture pure α^U.

Définition 2.7 (Transformation en UQ-ADT)**.** Soit un ADT $T = (\mathsf{A}, \mathsf{B}, \mathsf{Z}, \zeta_0, \tau, \delta) \in \mathcal{T}$. L'UQ-ADT correspondant est $T_{UQ} = (\mathsf{A}', \mathsf{B}', \mathsf{Z}, \zeta_0, \tau', \delta')$, avec :

$$
\begin{array}{rcl}
\mathsf{A}' & = & \hat{U}_T \cup \hat{Q}_T \cup \{\alpha^U, \alpha^Q : \alpha \in U_T \cap Q_T\} \\
\mathsf{B}' & = & \mathsf{B} \cup \{\bot\}
\end{array}
$$

$$
\tau' \quad : \quad \begin{cases}
\mathsf{Z} \times \mathsf{A}' & \to & \mathsf{Z} & \\
(\zeta, \alpha^U) & \mapsto & \tau(\zeta, \alpha) & \\
(\zeta, \alpha^Q) & \mapsto & \zeta & \\
(\zeta, \alpha) & \mapsto & \tau(\zeta, \alpha) & \text{si } \alpha \in \hat{U}_T \cup \hat{Q}_T
\end{cases}
$$

$$
\delta' \quad : \quad \begin{cases}
\mathsf{Z} \times \mathsf{A}' & \to & \mathsf{B}' & \\
(\zeta, \alpha^U) & \mapsto & \bot & \\
(\zeta, \alpha^Q) & \mapsto & \delta(\zeta, \alpha) & \\
(\zeta, \alpha) & \mapsto & \delta(\zeta, \alpha) & \text{si } \alpha \in \hat{U}_T \cup \hat{Q}_T
\end{cases}
$$

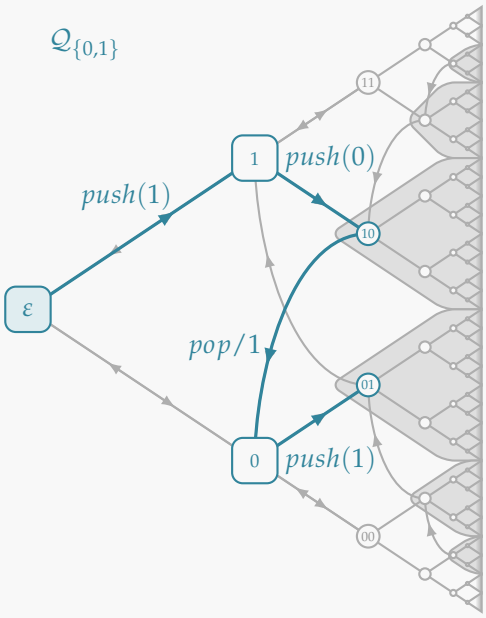
Dans un système séquentiel, utiliser une opération qui mêle écriture et lecture est équivalent à utiliser l'opération de lecture pure suivie de l'opération d'écriture pure correspondante, puisque ces deux opérations sont appliquées sur le même état. Dans un système réparti, l'équivalence n'est pas assurée : une opération concurrente pourrait s'intercaler entre la lecture et l'écriture. Les critères de cohérence faibles qui font l'objet de cette thèse ne peuvent pas garantir l'atomicité de l'opération. Séparer la partie lecture de la partie écriture peut donc permettre de redonner le contrôle de la concurrence à l'utilisateur de l'objet. Cela est illustré par l'exemple de la file dans la partie 4.2.

Composition d'ADT

Il est rare de n'utiliser qu'un seul objet dans un programme. Dans un système séquentiel, les histoires des différents objets s'entrelacent pour former une nouvelle histoire qui contient les opérations de tous les objets. Dans un système réparti, l'utilisation conjointe de plusieurs objets est plus compliquée à spécifier. Plutôt que de complexifier le modèle en envisageant l'utilisation d'un ensemble d'ADT, nous modélisons l'ensemble des objets comme une seule instance d'un ADT complexe composé des types abstraits de chacun des objets. Par exemple, un système de gestion de bases de don-

FILE

Une file *est une structure de données permettant de contenir des éléments d'un ensemble de valeurs X, triés selon une stratégie « premier entré, premier sorti » (*FIFO, *pour « first in, first out »). Elle est spécifiée par un type de données abstraits (définition 2.8) ayant deux sortes d'opérations. Un élément $x \in X$ peut être inséré en queue de file grâce à l'opération d'écriture $push(x)$. L'opération pop permet de récupérer le plus ancien élément encore présent et de l'enlever. Initialement, la file est vide. Nous choisissons la convention que, dans cet état où il n'est pas possible de lire la tête, la méthode pop ne modifie pas l'état et retourne la même valeur muette que $push(x)$.*
La file n'est pas un UQ-ADT car l'opération pop modifie l'état et retourne une valeur. La définition 2.9 exprime l'UQ-ADT correspondant, $\mathcal{Q}'_X$, qui sépare pop en une opération de lecture pure hd (pour « head ») qui retourne la tête de pile et une opération d'écriture pure $rh(x)$ (pour « remove head ») qui supprime la tête de pile si et seulement si celle-ci est égale à x.



Définition 2.8 (ADT file)**.** Soit X un ensemble dénombrable d'éléments et $\perp$ un élément n'appartenant pas à X. Le type $\mathcal{Q}_X$ est l'ADT :

$$\mathcal{Q}_X = (\mathsf{A} = \{push(x) : x \in X\} \cup \{pop\}, \mathsf{B} = \{\perp\} \cup X, \mathsf{Z} = X^\star, \zeta_0 = \varepsilon^2, \tau, \delta)$$

$$\tau : \begin{cases} \mathsf{Z} \times \mathsf{A} & \to & \mathsf{Z} \\ (\zeta, push(x)) & \mapsto & \zeta \cdot x \\ (x \cdot \zeta, pop) & \mapsto & \zeta \\ (\varepsilon, pop) & \mapsto & \varepsilon \end{cases} \qquad \delta : \begin{cases} \mathsf{Z} \times \mathsf{A} & \to & \mathsf{B} \\ (\zeta, push(x)) & \mapsto & \perp \\ (x \cdot \zeta, pop) & \mapsto & x \\ (\varepsilon, pop) & \mapsto & \perp \end{cases}$$

Définition 2.9 (UQ-ADT File)**.** Soit X un ensemble dénombrable d'éléments et $\perp$ un élément n'appartenant pas à X. Le type $\mathcal{Q}'_X$ est l'ADT :

$$\mathcal{Q}'_X = (\mathsf{A} = \{push(x), rh(x) : x \in X\} \cup \{hd\}, \mathsf{B} = \{\perp\} \cup X, \mathsf{Z} = X^\star, \zeta_0 = \varepsilon, \tau, \delta)$$

$$\tau : \begin{cases} \mathsf{Z} \times \mathsf{A} & \to & \mathsf{Z} & \\ (\zeta, push(x)) & \mapsto & \zeta \cdot x & \\ (x \cdot \zeta, rh(x)) & \mapsto & \zeta & \\ (y \cdot \zeta, rh(x)) & \mapsto & y \cdot \zeta & \text{si } x \neq y \\ (\varepsilon, rh(x)) & \mapsto & \varepsilon & \\ (\zeta, hd) & \mapsto & \zeta & \end{cases} \qquad \delta : \begin{cases} \mathsf{Z} \times \mathsf{A} & \to & \mathsf{B} \\ (\zeta, push(x)) & \mapsto & \perp \\ (\zeta, rh(x)) & \mapsto & \perp \\ (x \cdot \zeta, hd) & \mapsto & x \\ (\varepsilon, hd) & \mapsto & \perp \end{cases}$$

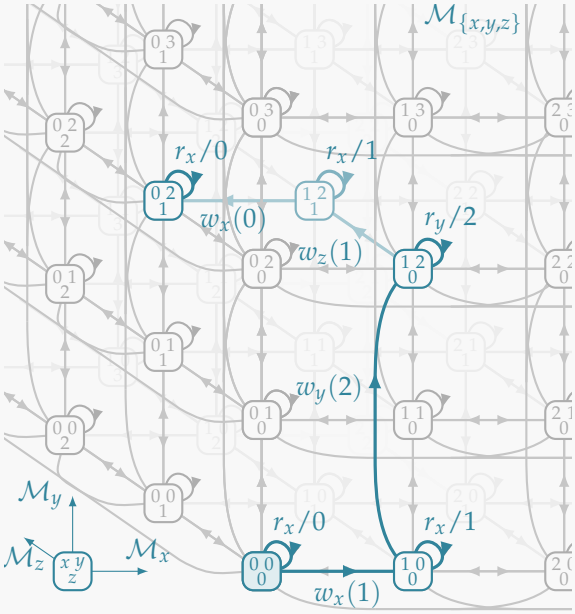
FIGURE 2.3 – La file.

nées ne donnera pas accès à plusieurs tables indépendantes, mais à une seule base de données composée de toutes les tables. De même, le type de données abstrait mémoire (figure 2.4) est une composition de registres. La composition de deux ADT T_1 et T_2 est le produit parallèle et asynchrone de leurs systèmes de transitions, formellement défini par la définition 2.12.

[2] ε désigne le mot de longueur 0.

REGISTRE ET MÉMOIRE

La mémoire partagée est certainement l'objet le plus étudié dans la littérature, car une branche importante de l'algorithmique du réparti concerne les modèles à mémoire partagée. Une mémoire $\mathcal{M}_X$ *est une collection de registres indépendants, chacun ayant un nom différent dans* X *(définition 2.11).*
Pour chaque $x \in X$, $\mathcal{M}_x$ *est le* registre *entier nommé* x*. Il possède une infinité d'opérations d'écriture de la forme* $w_x(n)$*, où* $n \in \mathbb{N}$ *est la valeur que l'on veut écrire. Sa seule opération de lecture,* r_x*, retourne la dernière valeur écrite, si elle existe, ou la valeur par défaut* 0 *sinon. Nous ne considérons ici que les registres contenant des nombres entiers et initialisés à* 0*, mais il serait facile d'étendre la définition 2.10 à d'autres types de données. Le registre est donc similaire au flux fenêtré de taille* 1*, seul le nom des opérations étant légèrement modifié.*



Définition 2.10 (Registre)**.** Soit x un *nom de registre*. Le *registre entier* sur x est l'UQ-ADT :

$$\mathcal{M}_x = (\mathsf{A} = \{\mathrm{w}_x(n) : n \in \mathbb{N}\} \cup \{\mathrm{r}_x\}, \mathsf{B} = \mathbb{N} \cup \{\bot\}, \mathsf{Z} = \mathbb{N}, \zeta_0 = 0, \tau, \delta)$$

$$\tau : \begin{cases} \mathsf{Z} \times \mathsf{A} & \to & \mathsf{Z} \\ (\zeta, \mathrm{w}_x(n)) & \mapsto & n \\ (\zeta, \mathrm{r}_x) & \mapsto & \zeta \end{cases} \qquad \delta : \begin{cases} \mathsf{Z} \times \mathsf{A} & \to & \mathsf{B} \\ (\zeta, \mathrm{w}_x(n)) & \mapsto & \bot \\ (\zeta, \mathrm{r}_x) & \mapsto & \zeta \end{cases}$$

Définition 2.11 (Mémoire)**.** Soit X un ensemble dénombrable de noms de registre. La mémoire entière sur X est la composition des registres de X, c'est-à-dire l'UQ-ADT :

$$\mathcal{M}_X = \prod_{x \in X} \mathcal{M}_x$$

FIGURE 2.4 – Le registre et la mémoire.

Définition 2.12 (Composition)**.** La composition de deux ADT $T = (\mathsf{A}, \mathsf{B}, \mathsf{Z}, \zeta_0, \tau, \delta)$ et $T' = (\mathsf{A}', \mathsf{B}', \mathsf{Z}', \zeta_0', \tau', \delta')$ est l'ADT $T \times T' = (\mathsf{A} \sqcup \mathsf{A}'$[3]$, \mathsf{B} \cup \mathsf{B}', \mathsf{Z} \times \mathsf{Z}', (\zeta_0, \zeta_0'), \tau'', \delta'')$:

$$\tau'' : \begin{cases} (\mathsf{Z} \times \mathsf{Z}') \times (\mathsf{A} \sqcup \mathsf{A}') & \to & \mathsf{Z} \times \mathsf{Z}' & \\ ((\zeta, \zeta'), \alpha) & \mapsto & (\tau(\zeta, \alpha), \zeta') & \text{si } \alpha \in \mathsf{A} \\ ((\zeta, \zeta'), \alpha') & \mapsto & (\zeta, \tau'(\zeta', \alpha')) & \text{si } \alpha' \in \mathsf{A}' \end{cases}$$

$$\delta'' : \begin{cases} (\mathsf{Z} \times \mathsf{Z}') \times (\mathsf{A} \sqcup \mathsf{A}') & \to & \mathsf{B} \cup \mathsf{B}' & \\ ((\zeta, \zeta'), \alpha) & \mapsto & \delta(\zeta, \alpha) & \text{si } \alpha \in \mathsf{A} \\ ((\zeta, \zeta'), \alpha') & \mapsto & \delta'(\zeta', \alpha') & \text{si } \alpha' \in \mathsf{A}'. \end{cases}$$

La composition est associative et commutative (à isomorphisme près) et $T_1 = (\emptyset, \emptyset, \{\zeta_0\}, \zeta_0, \emptyset, \emptyset)$ est un élément neutre. La structure $(\mathcal{T}, \times)$ est donc un monoïde commutatif. De plus, la composée de deux UQ-ADT est un UQ-ADT.

[3] $\sqcup$ désigne l'union disjointe (voir table des notations).

2.1.2 Histoires concurrentes

Nous avons vu dans la partie précédente que la spécification séquentielle d'un type de données abstrait est l'ensemble des histoires séquentielles qu'il admet. Il semble donc naturel que la spécification concurrente d'un objet partagé soit l'ensemble des histoires concurrentes qu'il admet. Une histoire concurrente est un modèle abstrait des exécutions réelles qui peuvent se produire. La façon de modéliser les exécutions est laissée au concepteur des systèmes et dépend du système considéré. De plus, les choix de modélisation (par exemple, la modélisation doit-elle tenir compte du temps réel ?) peuvent aboutir à des propriétés légèrement différentes. Nous présentons tout d'abord notre modèle d'histoires concurrentes, basé sur les structures asymétriques d'événements, puis nous l'appliquons au système qui nous intéresse particulièrement dans cette thèse, le cas des processus séquentiels communiquant par messages.

Définition

Il existe une grande diversité de modèles de systèmes répartis, allant des processus parallèles communicants aux réseaux pair-à-pair, en passant par des architectures dynamiques permettant la création de processus légers à la volée. Les objets partagés, qui sont justement utilisés dans un but d'abstraction de la complexité des systèmes sur lesquels ils sont implémentés, doivent avoir une spécification indépendante de ces systèmes.

Malgré leur grande diversité, tous ces systèmes ont un point commun : les événements qui s'y produisent ne sont pas totalement ordonnés. En général, les entités de calcul (processus, pairs, fils d'exécution, etc.) sont séquentielles, ce qui impose un ordre entre leurs propres événements, mais les événements concurrents ne sont pas ordonnés. Par exemple, dans le cas de processus séquentiels communicants, un événement a précède un événement b s'ils sont exécutés par le même processus dans cet ordre. Dans ce cas, chaque processus génère une chaîne maximale dans l'ordre partiel. Dans d'autres systèmes plus complexes, d'autres relations peuvent exister entre deux événements. Si deux processus p et q peuvent produire des événements a et b respectivement, tels que b provoque l'arrêt immédiat de p, on peut observer soit a suivi de b, soit b tout seul, mais pas b suivi de a. On dit alors que b préempte a.

Toutes ces notions sont en général capturées de manière compacte par les *structures asymétriques d'événements* [125]. Dans une structure asymétrique d'événements, les événements sont reliés par deux relations, traditionnellement nommées *causalité* et *causalité faible*. Pour éviter la confusion avec la cohérence causale et la réception causale, nous utiliserons dans ce manuscrit les dénominations *ordre de processus* et *ordre de programme*, employées dans certains travaux sur la cohérence [6,76]. Les structures asymétriques d'événements peuvent capturer le conflit : deux événements sont en conflit s'ils ne peuvent pas se produire dans la même exécution. Comme nous cherchons à modéliser les exécutions, encoder le conflit n'est pas nécessaire, ce qui nous permet de simplifier notre modèle d'*histoire concurrente* (définition 2.13). La figure 2.5 introduit la façon dont nous représenterons les histoires concurrentes graphiquement tout au long de cette thèse.

Remarque 2.2. Dans les systèmes répartis à passage de message, et en particulier dans les systèmes sans-attente qui forment le principal sujet d'étude de cette thèse, l'ordre de processus et l'ordre de programme sont confondus (voir partie 2.1.2). L'ordre de programme ne joue un rôle que dans le chapitre 7 et peut donc être ignoré jusque là.

Définition 2.13 (Histoire concurrente)**.** Une *histoire concurrente* est un quintuplet $H = (\Sigma, E, \Lambda, \mapsto, \nearrow)$ dont les différents champs sont détaillés ci-dessous.

- Σ est un ensemble dénombrable d'*opérations*.

  Dans notre modèle, les opérations sont typiquement celles d'un type de données abstrait (mais pas forcément exactement), de la forme α/β ou α.

- E est un ensemble dénombrable d'*événements*. Pour une histoire H, l'accesseur E_H permet d'accéder à l'ensemble des événements de H.

- $\Lambda : E \to \Sigma$ est la *fonction d'étiquetage* qui associe des opérations aux événements.

  On étend les notations des lectures (pures) et écritures (pures) de T aux événements de H : $\hat{U}_{T,H} = \{e \in E_H : \Lambda(e) \in \hat{U}_T\}$, et similairement pour $\hat{Q}_{T,H}$, $U_{T,H}$ et $Q_{T,H}$.

- $\mapsto$, l'*ordre de processus*, est une relation d'ordre sur E.

  Intuitivement, $e \mapsto e'$ signifie que e doit toujours se produire avant e'. Dans un système formé d'un ensemble de processus communicants, deux événements sont ordonnés par $\mapsto$ si, et seulement si, ils sont produits par le même processus. Chaque processus correspond donc à une chaîne maximale d'événements de H, c'est-à-dire un sous-ensemble maximal de E totalement ordonné par $\mapsto$. Dans la suite de cette thèse, nous noterons $\mathscr{P}_H$ l'ensemble des chaînes maximales de H et nous utiliserons le terme « processus » pour désigner une telle chaîne, même dans les modèles qui ne sont pas basés sur une collection de processus.

  On impose de plus à l'ordre de processus de vérifier les deux propriétés suivantes (on parle de *bel ordre*).

  Relation bien fondée. Il n'existe pas de suite infinie d'éléments de E strictement décroissante selon $\mapsto$. Dans notre modèle, cela signifie que l'exécution a une origine temporelle : il n'existe pas d'histoire qui existe depuis toujours, ni de processus infiniment rapide qui puisse produire une infinité d'événements en un temps fini.

  Absence d'antichaîne infinie. Dans tout sous-ensemble infini de E, il existe au moins un couple d'éléments ordonnés selon $\mapsto$. Dans notre modèle, cela signifie qu'à tout instant, le nombre d'entités dans le système est fini, bien que par forcément borné.

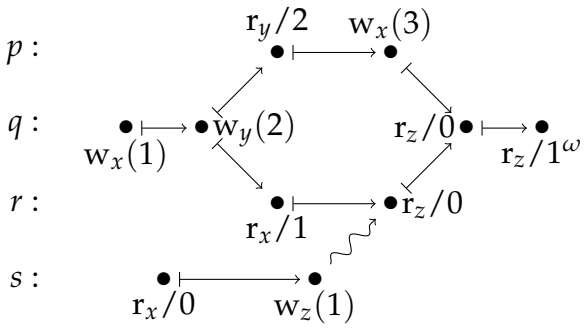
- $\nearrow$, l'*ordre de programme* est une relation d'ordre stricte (c'est-à-dire irréflexive) sur E telle que, pour tout couple $(e, e') \in E^2$ avec $e \neq e'$, si $e \mapsto e'$, alors $e \nearrow e'$.

  Intuitivement, $e \nearrow e'$ signifie que e ne peut jamais se produire après e', soit parce qu'ils sont reliés par l'ordre de processus, soit parce que e est préempté par e', c'est-à-dire que e' pourrait se produire avant e, mais il aurait comme action d'empêcher e de se produire ensuite. Formellement, e est préempté par e', noté $e \rightsquigarrow e'$, si $e \nearrow e'$ et $e \not\mapsto e'$.

Des systèmes répartis complexes peuvent être modélisés par les histoires concurrentes, comme nous le verrons dans le chapitre 7 dédié au langage d'orchestration Orc. Une histoire concurrente peut être vue comme une structure qui encode de manière concise plusieurs histoires séquentielles, chacune étant une linéarisation de l'histoire

```
1 fork
2   s:{r_x(); w_z(1); r_y();}
3   q:{
4     w_x(1); w_y(2);
5     fork
6       p:{w_x(1 + r_y());}
7       r:{r_x(); kill(s); r_z();}
8     while (true) {r_z();}
9   }
```

(a) Programme concurrent.



(b) Histoire correspondant à une exécution.

FIGURE 2.5 – Représentation graphique des histoires concurrentes. L'histoire concurrente de la figure 2.5b est un exemple d'exécution du programme de la figure 2.5a. Les points • représentent les événements qui se sont produits pendant l'exécution, étiquetés par les opérations opérations d'écriture et de lecture sur les registres partagés x, y et z. On ne représente pas la valeur de retour muette des écritures pures pour ne pas surcharger. Les événements sont reliés par deux sortes de flèches. Les flèches $\mapsto$ décrivent l'ordre de processus direct, dont l'ordre de processus est la fermeture transitive. À chaque appel de **fork**, Le processus se sépare en deux fils d'exécutions qui se rejoignent ensuite. L'histoire n'est donc pas linéaire. Les flèches $\rightsquigarrow$ décrivent la préemption : les événements de s ne peuvent pas se produire après l'événement $\mathrm{r}_z/0$, se produisant lui-même après l'appel de **kill** `(s)`. La notation ω en exposant désigne une chaîne infinie d'événements (selon $\mapsto$) étiquetés par la même opération.

concurrente (définition 2.14). Une linéarisation est donc une séquence d'opérations étiquetant des événements de l'histoire, dans un ordre qui respecte l'ordre de processus et l'ordre de programme.

Définition 2.14 (Linéarisation)**.** Soit $H = (\Sigma, E, \Lambda, \mapsto, \nearrow)$ une histoire concurrente. Une *linéarisation* de H est un mot (fini ou infini) $l = \Lambda(e_0) \dots \Lambda(e_n) \dots$ tel que

- les e_i sont des éléments distincts de E :

$$\{e_0, \dots, e_n, \dots\} \subset E \wedge \forall i \neq j, e_i \neq e_j$$

- l'ordre de programme est respecté :

$$\forall i, j, e_i \nearrow e_j \Rightarrow i < j$$

- l'ensemble des événements est fermé à gauche pour l'ordre de processus :

$$\forall e \in E, \forall e' \in \{e_0, \dots, e_n\}, e \mapsto e' \Rightarrow e \in \{e_0, \dots, e_n\}$$

- tous les événements qui ne sont pas préemptés sont représentés :

$$\forall e \in E, e \notin \{e_0, \dots, e_n\} \Rightarrow \exists e' \in \{e_0, \dots, e_n\}, e \nearrow e'$$

L'ensemble des linéarisations de H est noté $\mathrm{lin}(H)$.

Remarque 2.3. Le fait que l'ordre de processus est un bel ordre est important pour l'existence de linéarisations. Sans cette hypothèse, il serait possible qu'un événement ait un passé infini selon l'ordre de processus, auquel cas il serait impossible de lui donner une position finie dans la linéarisation. L'hypothèse du bel ordre sera à nouveau nécessaire dans le chapitre 5 pour prouver le théorème 5.10.

Nous présentons maintenant notre opérateur de projection, l'outil à tout faire qui va nous permettre de manipuler les histoires concurrentes pour définir les critères de cohérence. Soient $\dashrightarrow$ un bel ordre et A, B deux sous-ensembles d'événements. La projection $H^{\dashrightarrow}[\mathsf{A}/\mathsf{B}]$ conserve les écritures de A et les lectures de B et les réordonne selon $\dashrightarrow$. Plus précisément, elle a les trois actions suivantes.

- Elle supprime les événements qui ne sont pas dans A.
- Elle cache la valeur de retour des opérations qui ne sont pas dans B. Cela signifie que l'on ne considère la partie lecture que des opérations étiquetant des événements de B. En général, on fera en sorte que $\mathsf{A} \subset \mathsf{B}$. De toute façon, selon la définition 2.15, $H^{\dashrightarrow}[\mathsf{A}/\mathsf{B}] = H^{\dashrightarrow}[\mathsf{A} \cup \mathsf{B}/\mathsf{B}]$.
- Elle réordonne les événements dans l'histoire selon $\dashrightarrow$. Cela permet l'ajout ou la suppression de dépendances entre les événements. Si H ne contient pas de préemption et $\dashrightarrow$ est un bel ordre sur E_H, les événements de $H^{\dashrightarrow}[\mathsf{A}/\mathsf{B}]$ sont ordonnés selon $\dashrightarrow$. Sinon, pour éviter que des conflits ne soient créés entre l'ordre de processus et l'ordre de programme (qui correspondraient à des boucles dans l'union des deux ordres), ce dernier est également modifié. Si $\dashrightarrow$ est un ordre total, $H^{\dashrightarrow}[\mathsf{A}/\mathsf{B}]$ possède une et une seule linéarisation. Inversement, pour toute linéarisation l de H, il existe un ordre total $\dashrightarrow$ (unique en cas d'absence de préemption) tel que $\mathrm{lin}\,(H^{\dashrightarrow}[\mathsf{A}/\mathsf{B}]) = \{l\}$.

La projection est formellement définie dans la définition 2.15. La figure 2.6 illustre le fonctionnement de cette projection.

Définition 2.15 (Projection)**.** Soient $H = (\Sigma, E, \Lambda, \mapsto, \nearrow)$ une histoire concurrente, $\mathsf{A}, \mathsf{B} \subset E_H$ et $\dashrightarrow$ un bel ordre sur E_H. La *projection* $H^{\dashrightarrow}[\mathsf{A}/\mathsf{B}]$ est l'histoire concurrente $(\Sigma', E', \Lambda', \mapsto', \nearrow')$ dont les différents champs sont décrits ci-dessous.

- $E' = \mathsf{A} \cup \mathsf{B}$.
- Si $e \notin \mathsf{B}$ et $\Lambda(e) = \alpha/\beta$, alors $\Lambda'(e) = \alpha$. Sinon, $\Lambda'(e) = \Lambda(e)$.
- $\Sigma' = \Lambda'(E')$.
- $\mapsto'$ est la restriction de $\dashrightarrow$ à E'.
- Pour tout $e, e' \in E'$, $e \nearrow' e'$ si $e \neq e'$ et $e \dashrightarrow e'$ ou $e \rightsquigarrow e'$.

Quand les trois champs ne sont pas nécessaires, nous nous autorisons les notations simplifiées $H^{\dashrightarrow} = H^{\dashrightarrow}[E_H/E_H]$ et $H[\mathsf{A}/\mathsf{B}] = H^{\mapsto}[\mathsf{A}/\mathsf{B}]$.

Processus asynchrones communiquant par messages

Le but de cette partie est double. Il s'agit d'une part d'illustrer la définition des histoires concurrentes en les reliant à des systèmes répartis simples, et d'autre part de définir plus précisément les systèmes sans-attente, centraux dans la suite de cette thèse.

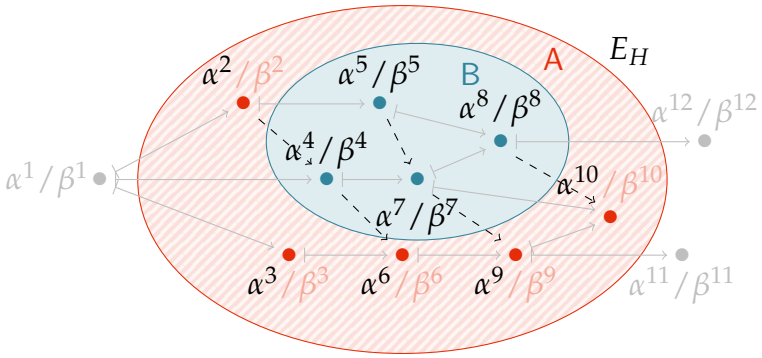


FIGURE 2.6 – L'histoire $H^{\dashrightarrow}[A/B]$ contient les écritures de $A \cup B$ et les lectures de B, ordonnées selon $\dashrightarrow$.

Définition du système. Le système de calcul est composé de n processus $p_0, ..., p_{n-1}$. Chaque processus s'exécute séquentiellement et possède une mémoire locale que l'on considère non bornée. Les processus sont asynchrones, c'est-à-dire qu'il n'existe pas de borne sur le temps d'exécution de leurs primitives élémentaires, ni relativement au temps d'exécution des primitives par les autres processus, ni relativement à celui des autres appels des primitives par le même processus.

Les processus communiquent en s'échangeant des messages grâce à un réseau complet de canaux de communication. Pour cela, les processus ont accès à des primitives de diffusion et de réception de messages. Un message diffusé est envoyé à tous les autres processus, y compris l'émetteur. Les canaux sont asynchrones, c'est-à-dire qu'il n'y a pas de borne sur le temps entre la diffusion et la réception d'un message. Un message diffusé peut par exemple être reçu très rapidement par un processus et au bout d'un temps très long par un autre. Lors de la diffusion d'un message, le processus émetteur le reçoit immédiatement (on peut imaginer que la primitive de diffusion appelle l'algorithme de réception d'un message localement).

Parmi les n processus, jusqu'à $t \leq n$ peuvent être *fautifs*. Un processus fautif arrête simplement d'opérer : à partir d'un certain point, appelé *panne franche*, il ne peut plus ni émettre ni recevoir de messages, ni exécuter d'actions. Une panne franche peut se produire au milieu d'une action, par exemple entre deux diffusions de message. Par contre, nous supposons que la diffusion est *uniforme*, c'est-à-dire que les messages sont acheminés à tous les processus même si l'émetteur tombe en panne au milieu d'une diffusion (voir plus bas). Cette hypothèse n'apporte pas de puissance de calcul au système l'uniformité peut être implémentée dans un système ne garantissant pas cette propriété [53]. Un processus qui n'est pas fautif est dit *correct*. Un processus correct, comme un processus fautif avant la panne, exécute normalement les algorithmes et la mémoire locale utilisée par l'algorithme ne peut pas être altérée en dehors de l'exécution de ceux-ci (même par une action locale).

Soit φ une condition ou un ensemble de conditions sur le nombre maximal t de fautes qui peuvent se produire pendant une exécution. On note $AS_n[\varphi]$ le système asynchrone à passage de messages composé de n processus dans lequel la condition φ est vérifiée. Dans $AS_n[t < \frac{n}{2}]$, il y a toujours une majorité de processus corrects dans le système, le système $AS_n[t \leq 1]$ autorise une faute dans le système et dans $AS_n[t = 0]$, aucune faute ne peut se produire.

Dans $AS_n[\emptyset]$, il n'y a pas de condition sur le nombre de fautes. Tous les processus sont donc susceptibles de tomber en panne pendant une exécution. Le système $AS_n[\emptyset]$

est appelé *système réparti asynchrone à passage de messages sans-attente*, ou plus simplement *système sans-attente*.

Un processus est soumis à deux sortes d'interaction : les appels aux opérations d'un type de données abstrait et les réceptions de messages. Le comportement des processus à ces instants-là est défini par un algorithme dépendant de l'objet implémenté. L'exécution de ces algorithmes est insécable : un algorithme ne peut pas recevoir de message d'un autre processus (rappelons que ses propres messages sont réceptionnés immédiatement) ni exécuter une opération de l'objet partagé lorsqu'il traite un autre message ou une autre opération.

Pour diffuser un message m, un processus dispose de trois primitives de communication : `broadcast`(m), `FIFO broadcast`(m) et `causal broadcast`(m) spécifiées par les propriétés suivantes.

1. La primitive `broadcast`, généralement appelée *diffusion fiable* [53], permet de répondre à la situation dans laquelle un émetteur tombe en panne pendant la diffusion et assure les deux propriétés suivantes.

   Validité. Si un processus reçoit un message m au moins une fois, alors m a été diffusé au moins autant de fois par un processus.

   Uniformité. Si un processus correct reçoit un message m, alors tous les processus corrects reçoivent m.

   Ces propriétés impliquent que tout message diffusé par un processus correct sera reçu par tous les processus corrects. En effet, si un processus correct diffuse un message, il le recevra localement immédiatement et comme il est correct, tous les autres processus corrects le recevront également d'après la propriété d'uniformité.

2. La primitive `broadcast` ne garantit aucun ordre sur les messages. La primitive `FIFO broadcast` assure les deux propriétés précédentes, ainsi qu'une propriété supplémentaire qui assure que deux messages diffusés par le même processus seront reçus dans l'ordre d'émission.

   Réception FIFO. Si un processus diffuse en utilisant `FIFO broadcast` un message m puis un message m', alors aucun processus ne recevra m' avant m,

3. Enfin, la primitive `causal broadcast` assure les deux propriétés de la primitive `broadcast` ainsi que le respect de la relation « s'est produit avant » de Lamport [71]. Notons que cette propriété implique celle de la primitive `FIFO broadcast`.

   Réception causale. Si un processus reçoit un message m puis diffuse un message m' en utilisant `causal broadcast`, alors aucun processus ne reçoit m' avant m.

En pratique, ces trois primitives peuvent être implémentées les unes à partir des autres dans $AS_n[\emptyset]$ [22, 98, 99, 106]. Elles n'apportent donc pas de puissance de calcul mais seulement du confort à l'utilisation. Par contre, elles n'ont pas le même coût. On préférera donc utiliser une primitive plus faible quand les hypothèses supplémentaires ne sont pas nécessaires.

Modélisation sous forme d'histoires concurrentes. L'exécution d'un programme utilisant un type de données abstrait $T = (\mathsf{A}, \mathsf{B}, \mathsf{Z}, \zeta_0, \tau, \delta)$ définit une histoire concurrente $H = (\Sigma, E, \Lambda, \mapsto, \nearrow)$ avec :

- l'ensemble des opérations est $\Sigma = \mathsf{A} \cup (\mathsf{A} \times \mathsf{B}')$, où B' contient B et toutes les valeurs de sortie qui peuvent être retournées par l'implémentation de T, y compris les messages d'erreur ;
- E contient l'ensemble des appels aux opérations de l'objet qui se sont produites et ont terminé lors de l'exécution.
- pour tout $e \in E$, $\Lambda(e) = \alpha/\beta$, où α est l'identifiant de la méthode appelée et β est la valeur retournée lors de l'exécution de e ;
- pour tout $e, e' \in E$, $e \nearrow e'$ si, et seulement si, e et e' sont des actions différentes du même processus et que e s'est produite avant e'. L'ordre de processus est la fermeture réflexive de l'ordre de programme.

Pour un algorithme A_T implémentant les actions liées aux opérations d'un type T et celles liées à la réception de tous les messages qu'il envoie, on note $\mathcal{H}(A_T)$ l'ensemble des histoires concurrentes qui peuvent être produites par un programme utilisant A_T comme implémentation de T.

2.1.3 Critères de cohérence

Le critère de cohérence est la propriété qui fait le lien entre une *spécification séquentielle* et les *histoires concurrentes* qu'elle autorise. De façon imagée, on peut voir un critère de cohérence comme un point de vue, une façon de regarder une histoire concurrente pour qu'elle paraisse séquentielle et en accord avec la spécification séquentielle.

L'ensemble des critères de cohérence

Comme la spécification séquentielle d'un objet dans un système séquentiel est l'ensemble des histoires séquentielles qu'il admet, la spécification concurrente d'un objet partagé est un ensemble d'histoires concurrentes. Un critère de cohérence caractérise les histoires concurrentes admissibles pour un ADT donné. C'est donc une fonction qui associe une spécification concurrente à chaque ADT (définition 2.17).

Définition 2.17 (Critère de cohérence)**.** Un critère de cohérence est une fonction

$$C : \mathcal{T} \to \mathcal{P}(\mathcal{H})$$

L'ensemble des critères de cohérence est noté $\mathcal{C}$.

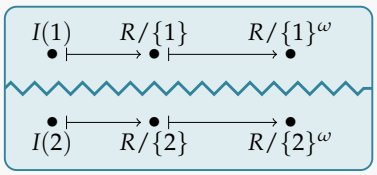
Un algorithme A_T est C-cohérent pour un critère $C \in \mathcal{C}$ et un ADT $T \in \mathcal{T}$ si toutes ses opérations terminent et toutes les exécutions qu'il admet sont C-cohérentes, c'est-à-dire si elles appartiennent à $C(T)$.

La figure 2.7 illustre notre modèle en définissant formellement la cohérence locale. La cohérence locale (*LC*, pour « local consistency ») est le critère de cohérence des variables locales, qui ne sont pas partagées entre les processus : chaque processus applique ses écritures sur sa copie locale et aucune donnée n'est échangée entre les processus.

Cohérence Locale

✓ Composable p. 41
✓ Décomposable p. 41
✗ Non partagé p. 126

Une variable locale contient une instance d'objet qui n'est pas partagé. Les appels aux opérations de cette instance forment tout de même des histoires concurrentes qui ont une structure particulière, spécifiée par un critère de cohérence : la cohérence locale. Dans la cohérence locale, chaque processus est indépendant des autres. Par exemple, dans l'histoire ci-contre, deux processus écrivent et lisent dans un ensemble d'entiers (voir page 27). Le premier processus insère le nombre 1 puis lit l'état $\{1\}$ une infinité de fois. Le deuxième processus insère le nombre 2 puis lit l'état $\{2\}$ une infinité de fois. Toutes les lectures contiennent bien les valeurs insérées par le processus qui fait la lecture, indépendamment de ce que fait l'autre processus.



Formellement, une histoire H est localement cohérente pour un type de données abstrait T (définition 2.16) si, pour chaque processus $p \in \mathscr{P}_H$, la linéarisation de l'histoire $H[p/p]$, contenant uniquement les événements de p (qui est unique car ses événements sont totalement ordonnés), est conforme à la spécification séquentielle $L(T)$ de T. Dans l'histoire ci-dessus, les linéarisations suivantes des événements de chaque processus sont cohérentes avec la spécification séquentielle de l'objet partagé :

$$I(1) \cdot R/\{1\}^\omega \qquad\qquad I(2) \cdot R/\{2\}^\omega$$

Définition 2.16 (Cohérence locale)**.** La *cohérence locale* est le critère de cohérence :

$$LC : \begin{cases} \mathcal{T} & \to & \mathcal{P}(\mathcal{H}) \\ T & \mapsto & \{H \in \mathcal{H} : \forall p \in \mathscr{P}_H, \mathrm{lin}(H[p/p]) \cap L(T) \neq \emptyset\} \end{cases}$$

Le seul changement de l'histoire ci-contre par rapport à l'histoire localement cohérente ci-dessus est la valeur retournée par les lectures du deuxième processus à partir de la deuxième lecture. Le deuxième processus lit maintenant $\{2\}$ puis $\{1,2\}$ sans avoir inséré 1 entre ses deux lectures. L'histoire n'est pas localement cohérente car, dans la linéarisation pour le deuxième processus, le passage de l'état $\{2\}$ à l'état $\{1,2\}$ n'est pas conforme à la spécification séquentielle.

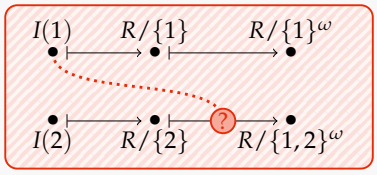


FIGURE 2.7 – La cohérence locale.

L'espace des critères de cohérence hérite naturellement des propriétés de l'ensemble des parties d'un ensemble, et en particulier de sa structure de treillis pour l'inclusion. L'ordre $\leq$ exprime la force du critère de cohérence. Si $C_1 \leq C_2$, alors C_2 est *plus fort* que C_1, car il garantit de plus fortes propriétés sur les histoires qu'il admet. Ainsi, une implémentation C_2-cohérente peut toujours être utilisée à la place d'une implémentation C_1-cohérente du même ADT si $C_1 \leq C_2$. La borne supérieure, donnée par l'opérateur $+$, est la *conjonction*. Elle permet de renforcer des critères de cohérence en les utilisant conjointement, de façon à garantir les propriétés des différents critères en même temps.

Définition 2.18 (Treillis des critères de cohérence)**.** Soit $\leq$ la relation définie sur les critères de cohérence par $C_2 \leq C_1$ si $\forall T \in \mathcal{T}, C1(T) \subset C2(T)$.

La structure algébrique $(\mathcal{C}, \leq)$ est un treillis borné dont les deux lois sont $+$ (borne supérieure) et $\triangle$ (borne inférieure), le maximum est $C_\top$ et le minimum $C_\bot$, définis par :

$$C_1 + C_2 : \begin{cases} \mathcal{T} & \to & \mathcal{P}(\mathcal{H}) \\ T & \mapsto & C_1(T) \cap C_2(T) \end{cases} \qquad C_1 \triangle C_2 : \begin{cases} \mathcal{T} & \to & \mathcal{P}(\mathcal{H}) \\ T & \mapsto & C_1(T) \cup C_2(T) \end{cases}$$

$$C_\top : \begin{cases} \mathcal{T} & \to & \mathcal{P}(\mathcal{H}) \\ T & \mapsto & \emptyset \end{cases} \qquad C_\bot : \begin{cases} \mathcal{T} & \to & \mathcal{P}(\mathcal{H}) \\ T & \mapsto & \mathcal{H} \end{cases}$$

Composition des critères de cohérence

Les programmes complexes sont en fait souvent composés de plusieurs objets partagés. D'ailleurs, le principal intérêt de la programmation orientée objets est de pouvoir encapsuler des briques logicielles dans des classes isolées les unes des autres. Une question légitime est donc l'étude du comportement conjoint d'objets partagés, s'ils sont individuellement cohérents. Plus spécialement, on considère un programme composé d'instances de deux types de données abstraits T_1 et T_2 qui vérifient le même critère de cohérence C.

Le comportement espéré est que les deux objets se comportent conjointement comme s'il n'y avait qu'une seule instance du type composé $T_1 \times T_2$ avec le critère de cohérence C. Cette propriété est appelée *localité* (définition 2.19). La localité se décompose en deux propriétés. La composabilité assure que si deux objets sont cohérents, leur composition est également cohérente. Inversement, la décomposabilité assure que si la composition est cohérente, les composants sont eux-mêmes cohérents.

La cohérence locale est un critère local. La similitude de dénomination entre la cohérence locale et la localité ne se réfère pas à la même notion : la « localité » du critère de cohérence est spatiale (par rapport au processus sur lequel est appelée l'opération) alors que celle de la propriété est plutôt temporelle : le terme a été initialement utilisé dans [121] pour décrire la linéarisabilité (voir page 47) qui tire sa composabilité de son ancrage dans le temps réel. La localité est une propriété très difficile à obtenir, comme nous le verrons dans la partie 2.2, et en dehors de la cohérence locale et de la linéarisabilité, les seuls critères locaux que nous présenterons dans cette thèse sont très faibles.

Définition 2.19 (Composabilité, décomposabilité et localité)**.** Soit $C(T_1, T_2)$ l'ensemble des histoires dont les projections sur les événements concernant $T_1 \in \mathcal{T}$ et $T_2 \in \mathcal{T}$ sont toutes les deux C-cohérentes. Formellement, une histoire H est dans $C(T_1, T_2)$ s'il existe une partition des événements $\{E_1, E_2\}$ des événements de E_H telle que les projections de H sur E_1 et E_2 sont C-cohérentes pour chacun des deux ADT.

$$C(T_1, T_2) = \left\{ \begin{array}{rl} H \in \mathcal{H} : \exists E_1, E_2 \subset E_H, & E_1 \sqcup E_2 = E_H \\ \wedge & H[E_1/E_1] \in C(T_1) \\ \wedge & H[E_2/E_2] \in C(T_2) \end{array} \right\}$$

Un critère $C \in \mathcal{C}$ est *composable* si : $\forall T_1, T_2 \in \mathcal{T}, C(T_1 \times T_2) \supset C(T_1, T_2)$.
Un critère $C \in \mathcal{C}$ est *décomposable* si : $\forall T_1, T_2 \in \mathcal{T}, C(T_1 \times T_2) \subset C(T_1, T_2)$.
Un critère $C \in \mathcal{C}$ est *local* si : $\forall T_1, T_2 \in \mathcal{T}, C(T_1 \times T_2) = C(T_1, T_2)$.

Fermeture par composition. Que se passe-t-il si l'on utilise tout de même dans un même programme des objets partagés C-cohérents pour des types de données abstraits T_1 et T_2, alors que C n'est pas composable ? Les histoires obtenues ne seront pas forcément C-cohérentes pour l'ADT $T_1 \times T_2$, mais elles vérifieront tout de même la propriété suivante : leurs projections sur les événements de T_1 et de T_2 seront C-cohérentes. Le critère de cohérence qui modélise ces histoires est appelé la *fermeture par composition* de C, noté $C^\star$. Plus précisément, la définition 2.20 définit la fermeture par composition de C comme la borne supérieure de tous les critères composables plus faibles que C et la proposition 2.1 montre qu'il s'agit du plus fort critère composable plus faible que C.

Définition 2.20 (Fermeture par composition)**.** Soit C un critère de cohérence. La fermeture par composition de C est le critère de cohérence :

$$C^\star = \sum \{C' \leq C : C' \text{ composable}\}$$

Proposition 2.1. *Pour tout C, $C^\star$ est le plus fort critère composable plus faible que C.*

Démonstration. Premièrement, $C^\star \leq C$. En effet, soient $T \in \mathcal{T}$ et $H \in C(T)$. Pour tout critère $C' \leq C$ composable, $H \in C(T)$, donc $H \in C'(T)$. Ainsi, $H \in C^\star(T)$, donc $C^\star \leq C$.

Deuxièmement, $C^\star$ est composable. En effet, soient $T_1, T_2 \in \mathcal{T}$, $H_1 \in C^\star(T_1)$, $H_2 \in C^\star(T_2)$ et $H \in H_1 \times H_2$. Pour tout $C' \leq C$ composable, $H_1 \in C'(T_1)$, $H_2 \in C'(T_2)$. Or C' est composable, donc $H \in C'(T_1 \times T_2)$. Finalement, $H \in C^\star(T_1 \times T_2)$ donc $C^\star$ est composable.

Troisièmement, $C^\star$ est plus fort que tous les autres critères composables plus faibles que C, par construction. □

2.2 Confrontation à l'existant

Dans cette section, nous étudions les critères de cohérence existant dans la littérature et nous cherchons à les exprimer dans notre formalisme. Nous avons fait attention à ce que les critères définis ici coïncident avec leurs définitions de l'état de l'art. Par exemple, la cohérence pipeline, appliquée à la mémoire dans $AS_n[\emptyset]$ autorise les mêmes histoires que la mémoire PRAM de [76]. D'autres extensions pourraient être proposées. En particulier, nous ne tenons pas compte de l'ordre de programme car il est égal à l'ordre de processus dans $AS_n[\emptyset]$.

Nous définissons formellement les critères les plus importants dans notre modèle au sein d'une fiche illustrée à l'aide des histoires de la figure 2.8. Ces critères sont représentés sur la figure 2.8a. Les zones en rouge hachuré sur cette figure ne contiennent que des critères plus faibles qu'un critère C non composable mais pas que sa fermeture par composition $C^\star$. D'après la proposition 2.1, ces critères ne sont donc pas composables.

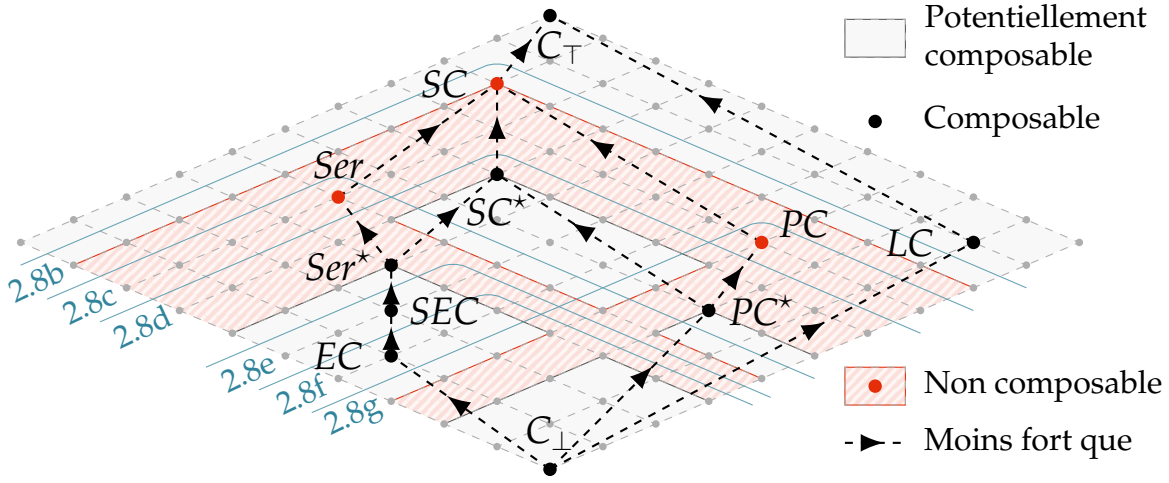
2.2.1 La cohérence forte

La cohérence séquentielle

Le but de la cohérence séquentielle (*SC*, pour « sequential consistency ») [72] est de mimer l'existence d'une unique copie de l'objet dans le réseau, à laquelle tous les processus accèdent séquentiellement. Une façon de se la représenter est d'imaginer que cette copie partagée est stockée sur un serveur, comme sur la figure 2.9a. Pour écrire, un processus envoie simplement un message au serveur. Pour lire, il envoie un message au serveur et attend sa réponse. Comme le serveur traite les messages séquentiellement, les opérations sur l'objet partagé apparaîtrons totalement ordonnées. Notons bien que le serveur n'est pas nécessaire à l'implémentation de la cohérence séquentielle ; un objet partagé séquentiellement cohérent doit seulement *se comporter* de la même façon.

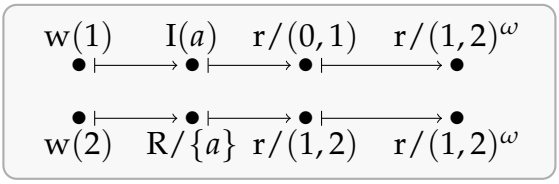
Plus précisément, une histoire concurrente est séquentiellement cohérente s'il existe une histoire séquentielle telle que :

- l'histoire séquentielle contient les mêmes événements que l'histoire concurrente,
- deux événements du même processus apparaissent dans le même ordre dans l'histoire séquentielle,
- l'histoire séquentielle est correcte vis-à-vis de l'objet considéré.

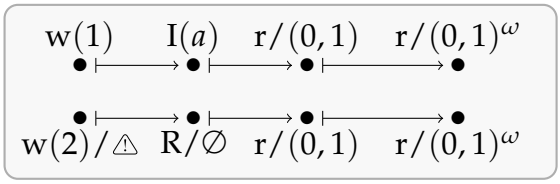


$SC(SC^\star)$	$Ser(Ser^\star)$	$PC(PC^\star)$	$EC(SEC)$	LC	$C_\top$ $C_\bot$
Cohérence séquentielle (de cache)	Sérialisabilité (fermeture par composition)	Cohérence pipeline (mémoire lente)	Convergence (forte)	Cohérence locale	Critères maximal et minimal
(p. 46, 63)	(p. 52, 41)	(p. 62, 41)	(p. 54, 56)	(p. 40)	(p. 40)

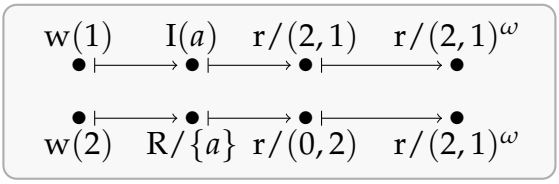
(a) Treillis des critères de cohérence.



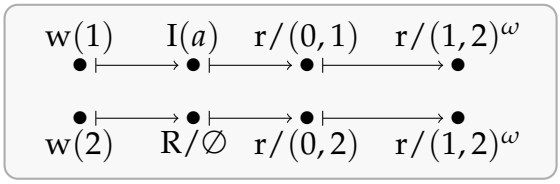
(b) *SC*



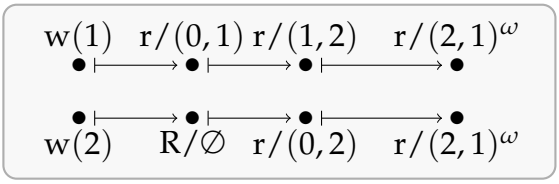
(c) *Ser* mais pas $SC^\star$ ni $PC^\star$



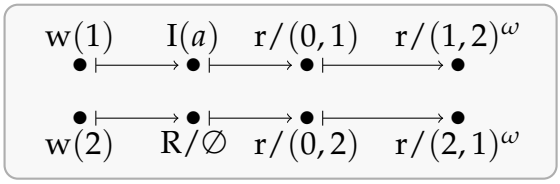
(d) $SC^\star$ mais pas *Ser* ni *PC*



(e) *SEC* mais pas $Ser^\star$ ni $PC^\star$

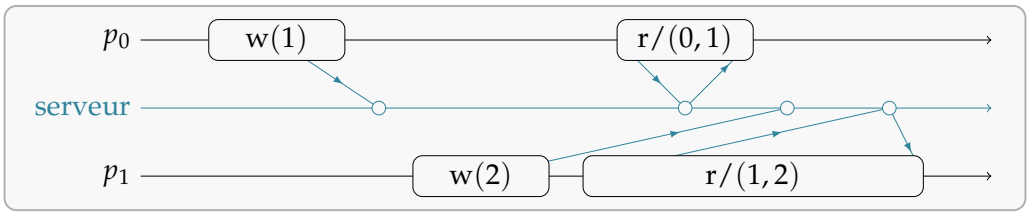


(f) *EC* mais pas *SEC* ni $PC^\star$

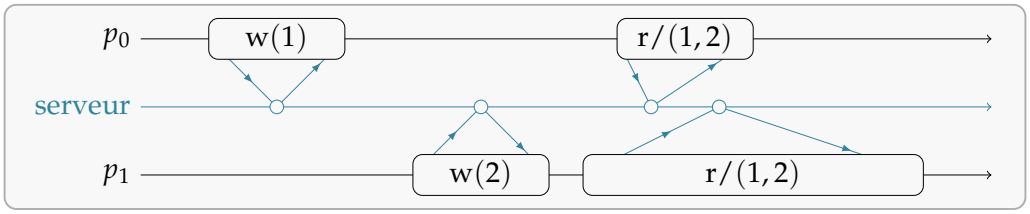


(g) *PC* mais pas *EC*

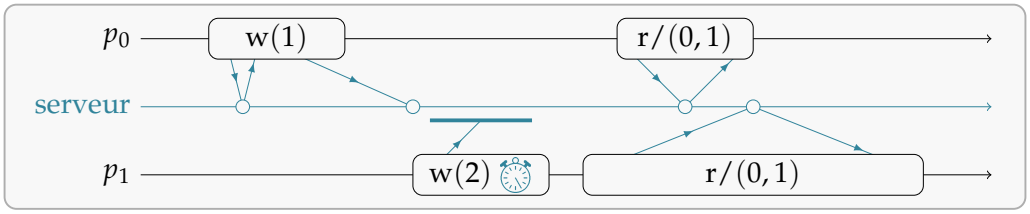
FIGURE 2.8 – Pour chaque histoire (sauf celle de la figure 2.8f), une instance du type $\mathcal{W}_2 \times \mathcal{S}_{[a-z]}$, c'est-à-dire la composition d'un flux fenêtré de taille 2 (définition 2.5) et d'un ensemble de lettres (définition 2.2) est partagée entre deux processus ; le premier processus écrit 1 dans le flux fenêtré (opération $\mathrm{w}(1)$) puis insère a dans l'ensemble (opération $\mathrm{I}(a)$), pendant que le deuxième processus écrit 2 dans le flux fenêtré (opération $\mathrm{w}(2)$) puis lit l'ensemble (opération R). Ensuite tous deux lisent le flux fenêtré une infinité de fois (opération r). Sur la figure 2.8f, l'insertion dans l'ensemble est remplacée par une lecture du flux fenêtré.



(a) La cohérence séquentielle.



(b) La linéarisabilité.



(c) La sérialisabilité.

FIGURE 2.9 – La cohérence forte vise à donner l'impression de l'existence d'une unique copie de l'objet partagée à laquelle tous les processus accèdent séquentiellement. Dans ces histoires, deux processus accèdent à un flux fenêtré de taille 2 en communiquant avec un serveur qui gère l'unique copie de l'objet. Le processus p_0 écrit 1 dans le flux fenêtré (opération w(1)), le deuxième processus écrit 2, puis ils lisent tous les deux (opération r).

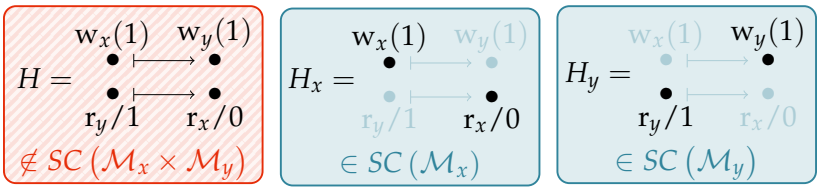


FIGURE 2.10 – La cohérence séquentielle n'est pas composable. Les deux processus de l'histoire H partagent une instance de mémoire $\mathcal{M}_{\{x,y\}} = \mathcal{M}_x \times \mathcal{M}_y$ et les histoires H_x et H_y sont projections de H qui ne conservent que les événements concernant les registres $\mathcal{M}_x$ et $\mathcal{M}_y$ respectivement.

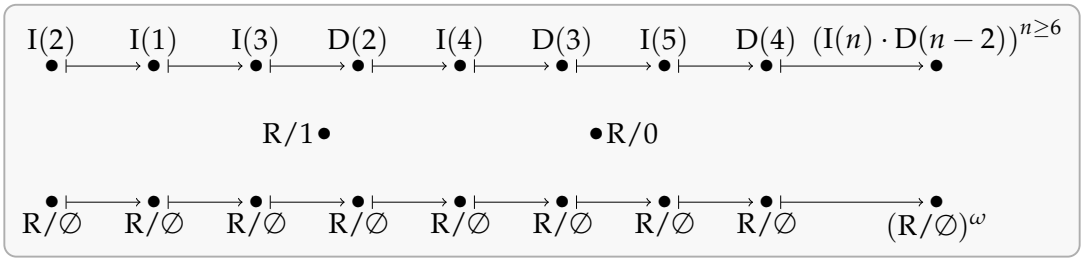


FIGURE 2.11 – Un processus insère (opération I(n)) et supprime (opération D(n)) des entiers dans un ensemble. Les autres lisent (opération R). Quels lectures sont possibles selon la cohérence séquentielle ?

La cohérence séquentielle n'est pas composable, comme le montre la figure 2.10. Deux processus partagent une instance de mémoire composée de deux registres x et y. Le premier processus écrit 1 dans x puis 1 dans y. Le deuxième processus lit 1 dans y puis 0 dans x. Si l'on étudie chaque registre indépendamment, les deux sous-histoires sont séquentiellement cohérentes : la lecture de x a été faite avant son écriture et la lecture de y suit son écriture. Pourtant, l'union de ces deux ordres et de l'ordre de processus contient un cycle donc l'histoire complète n'est pas séquentiellement cohérente.

La définition originale de la cohérence séquentielle par Lamport [72] peut s'appliquer à tout type de données abstrait. Sur la figure 2.12, nous nous contentons de transcrire cette définition dans notre modèle.

Cohérence séquentielle et histoires infinies. Intéressons-nous à l'histoire de la figure 2.11, et plus particulièrement aux deux processus qui font une infinité d'opérations. Le premier processus insère les valeurs 2, 1 puis 3 dans un ensemble partagé séquentiellement cohérent puis alterne une suppression du plus petit élément différent de 1 de l'ensemble et l'insertion du plus petit élément supérieur ou égal à 1 qui n'a pas encore été inséré. Un autre processus peut-il lire indéfiniment la valeur $\emptyset$?

Remarquons tout d'abord que la première lecture peut très bien retourner $\emptyset$. Il suffit pour cela qu'elle soit placée avant la première écriture dans la linéarisation. Il en va de même pour la deuxième, la troisième, et ainsi de suite. Chaque lecture peut donc retourner $\emptyset$, mais toutes le peuvent-elles ? Pour faire cela, la première écriture devrait être placée après une infinité de lectures. Cela est-il autorisé ?

La définition de la cohérence séquentielle telle qu'elle est définie dans [72] n'est en fait pas assez précise pour répondre à cette question et aucun consensus clair ne ressort au sein de la communauté scientifique sur l'interprétation la plus adéquate. Cependant, les deux interprétations possibles mènent à des critères très différents. Dans [110], Sezguin montre que plusieurs objets dont la pile et la file peuvent être implémentés de manière séquentiellement cohérente sans échanger un seul message si le point de linéarisation peut être reporté à l'infini. À l'inverse, dans [14], Attiya et Welch montrent que même dans un système synchrone, un envoi de message et une attente de réponse sont nécessaires pour au moins une opération (empiler ou dépiler). L'interprétation d'Attiya et Welch est donc considérablement plus forte que l'autre.

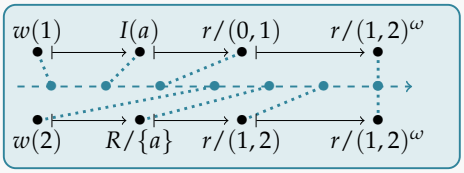
Cet exemple prouve l'importance de la précision dans la modélisation des objets partagés. Premièrement, une définition précise permet d'éviter ce genre de débat ; à un détail près, un objet partagé peut garantir des propriétés très fortes ou au contraire très faibles. Deuxièmement, les difficultés rencontrées pendant la spécification sont un bon indicateur des limites d'un modèle. Dans ce cas précis, si l'on peut placer une écriture après une infinité de lectures, pourquoi ne pourrait-on pas placer une lecture après une infinité d'écritures ? Un processus peut-il lire l'état $\{1\}$ correspondant à l'ensemble des valeurs insérées et jamais supprimées par le premier processus, bien que cet état ne soit jamais atteint ? L'application successive d'une infinité d'écritures est appelée une *hypertâche* (« hypertask » en anglais) et toutes les hypertâches n'ont pas de limite. Peut-il lire n'importe quel état, par exemple 0, même si la valeur 0 n'est jamais insérée, puisque le chemin dans le système de transitions ne spécifiera jamais une lecture faite après une infinité d'opérations ?

Nous pensons que ce genre de questions est surtout un signe que la deuxième interprétation n'est pas adaptée pour formaliser l'intuition que l'on a du critère de cohérence. La version forte de la cohérence séquentielle semble plus naturelle. C'est celle que l'on considère dans toute la suite de la thèse. Cela est garanti par la définition des linéarisations selon laquelle chaque événement est placé à une position finie.

COHÉRENCE SÉQUENTIELLE

✗ Non composable p. 41
✓ Décomposable p. 41
✗ Fort p. 126

La cohérence séquentielle fut initialement définie par Lamport dans [72] comme : « le résultat de n'importe quelle exécution est le même que si les opérations de tous les processus avaient été exécutées dans un ordre séquentiel, et les opérations de chaque processus apparaissent dans cette séquence dans l'ordre spécifié par son programme ». C'est par exemple le cas dans l'histoire de la figure 2.8b (rappelée ci-contre) : si l'on ordonne tous les événements selon la ligne bleue, le flux fenêtré passe de l'état $(0,0)$ à l'état $(0,1)$ lors de l'écriture $w(1)$ et de l'état $(0,1)$ à l'état $(1,2)$ lors de l'écriture $w(2)$, ce qui est conforme à sa spécification séquentielle, et de même pour l'ensemble.



Formellement, une histoire H est séquentiellement cohérente par rapport à un type de données abstrait T (définition 2.21) s'il existe une séquence qui possède les deux propriétés suivantes.

- *Elle contient les étiquettes de tous les événements de H et l'ordre dans lequel elles apparaissent est compatible avec l'ordre des processus, c'est-à-dire qu'elle appartient à $\mathrm{lin}(H)$.*
- *Elle est correcte par rapport à la spécification séquentielle de l'objet, c'est-à-dire qu'elle appartient à $L(T)$.*

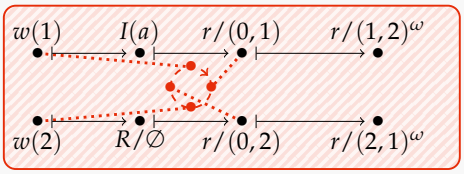
Dans l'exemple ci-dessus, la linéarisation suivante correspond à ces critères :

$$w(1)/\bot \cdot I(a)/\bot \cdot r/(0,1) \cdot w(2)/\bot \cdot R/\{a\} \cdot r/(1,2)^\omega$$

Définition 2.21 (Cohérence séquentielle)**.** La *cohérence séquentielle* est le critère de cohérence :

$$SC : \begin{cases} \mathcal{T} & \to & \mathcal{P}(\mathcal{H}) \\ T & \mapsto & \left\{ H \in \mathcal{H} : \mathrm{lin}(H) \cap L(T) \neq \emptyset \right\} \end{cases}$$

Les autres histoires de la figure 2.8 ne sont pas séquentiellement cohérentes. Essayer d'ordonner tous les événements d'une histoire qui n'est pas séquentiellement cohérente résulte généralement en un cycle. Un bon exemple est l'histoire de la figure 2.8g : une linéarisation qui place l'écriture $w(1)$ avant la lecture $r/(0,2)$ ne respecte pas la spécification séquentielle, donc une linéarisation doit placer $r/(0,2)$ avant $w(1)$ – et de la même manière $r/(0,1)$ avant $w(2)$ – ce qui est impossible sans violer l'ordre de processus. On voit ici ce qui rend la cohérence séquentielle difficile à implémenter : les deux processus doivent forcément se synchroniser pour briser ce cycle.



Remarquons que l'histoire de la figure 2.8c (à droite) n'est pas non plus séquentiellement cohérente. En effet, bien qu'il soit possible d'établir un ordre total sur tous les événements et qu'ainsi ordonnés, la spécification séquentielle est respectée (car tout préfixe fini la respecte), cet ordre total ne décrit pas une linéarisation valide car les événements du deuxième processus ont une infinité de prédécesseurs dans l'ordre total.

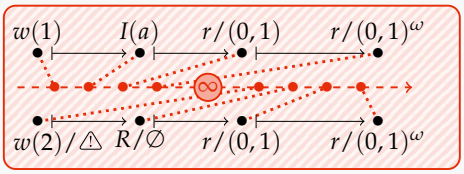


FIGURE 2.12 – La cohérence séquentielle.

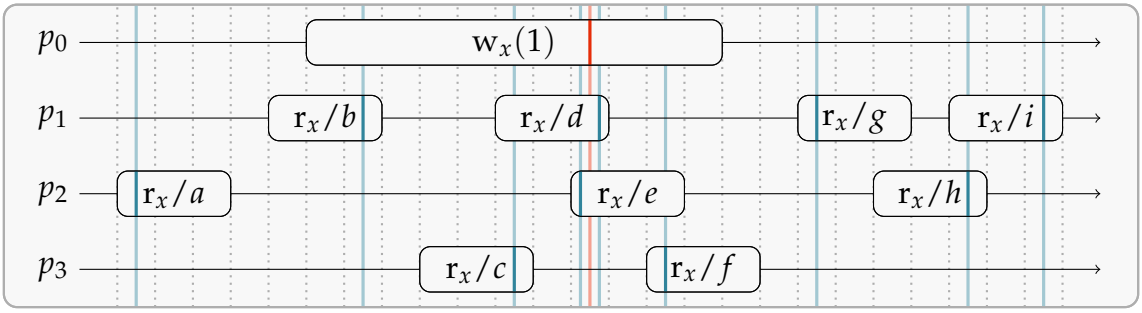


FIGURE 2.13 – Histoire répartie montrant un registre partagé entre quatre processus. Avec les points de linéarisation représentés en bleu, on doit avoir $a = b = c = e = 0$ et $d = f = g = h = i = 1$ pour la linéarisabilité.

Linéarisabilité

En 1986, Lamport propose trois spécifications du registre partagé : le registre sûr, le registre régulier et le registre atomique [74]. La particularité de son modèle est que la concurrence est définie en référence au temps réel. Plus précisément, chaque événement a un *début* situé à l'instant où l'opération est appelée et une *fin* située à l'instant où l'opération termine. Deux opérations sont concurrentes si leurs exécutions se chevauchent dans le temps, c'est-à-dire si la fin de l'une n'est pas située avant le début de l'autre. Lamport simplifie encore le problème en supposant qu'un seul processus est autorisé à écrire dans le registre, les autres ne pouvant que lire. Cela permet d'instaurer un ordre naturel sur les écritures et donc de pouvoir dire qu'une écriture est plus récente qu'une autre. Des lectures peuvent toujours être concurrentes avec des écritures ou avec d'autres lectures. La figure 2.13 présente une histoire concurrente dans laquelle un registre est partagé par quatre processus. Seul le processus p_0 peut écrire dans le registre.

Registre sûr. La sûreté garantit que si une lecture n'est concurrente avec aucune écriture, alors la valeur lue doit être la dernière valeur écrite, ou la valeur initiale si aucune écriture n'a encore eu lieu. Sur la figure 2.13, $a = 0$ et $g = h = i = 1$. Les lectures concurrentes avec une écriture ne sont pas spécifiées : elles peuvent retourner n'importe quelle valeur acceptée par le type stocké dans le registre. L'exemple qui illustre le mieux le registre sûr est le tableau à volets affichant les départs dans certaines gares : les mises à jour prennent un certain temps et si l'on prend une photographie pendant l'édition, il est possible de voir des informations complètement incohérentes. Sur la figure 2.14, les villes visitées par le Thalys de 12 h 55 après Bruxelles sont illisibles.

Registre régulier. En plus de la sûreté, la régularité impose aux lectures concurrentes avec une écriture de retourner soit la dernière valeur dont l'écriture est terminée soit la valeur en cours d'écriture. Sur la figure 2.13, b, c, d, e et f valent toutes soit 0 soit 1. Par contre, rien n'oblige les lectures successives du même processus à voir des valeurs de plus en plus récentes. Par exemple, il est possible d'avoir $b = 1$ et $d = 0$.

Registre atomique. L'atomicité pallie ce problème en imposant que si deux lectures ne sont pas concurrentes, la deuxième doit obligatoirement retourner une valeur au moins aussi récente que la première. Sur la figure 2.13, on peut avoir $d = 1$ et $e = 0$ car les lectures qui retournent d et e sont concurrentes, mais cela impose $b = c = e = 0$ et $d = f = 1$ car les écritures qui retournent b et c précèdent dans le temps celle qui retourne e et celle qui retourne f suit celle qui retourne d.

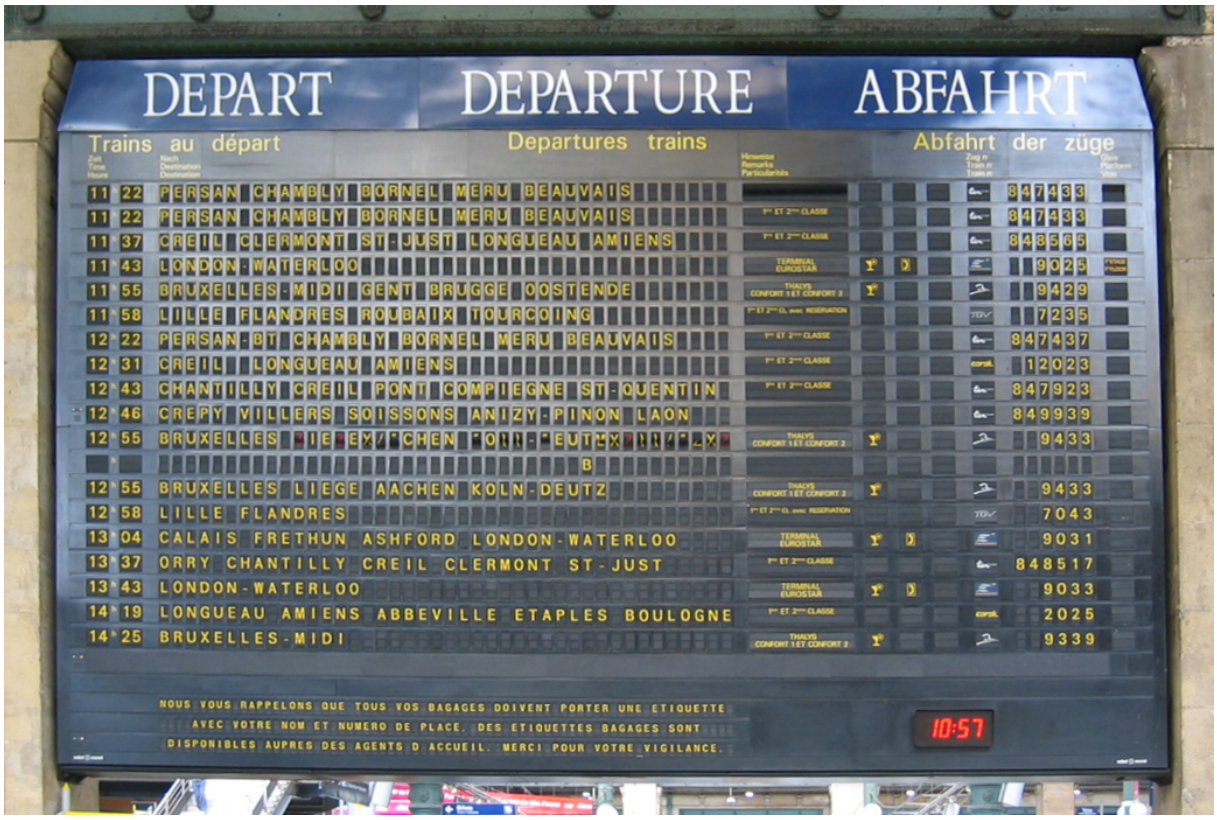


FIGURE 2.14 – Ancien tableau des départs de grandes lignes, en Gare du Nord à Paris[4].

Quatre ans plus tard, en 1990, l'atomicité a été étendue aux autres types de données abstraits par Herlihy [58] sous le nom de *linéarisabilité*. Une histoire H est linéarisable pour un type de données abstrait T si, pour tout événement de H, il existe un *point de linéarisation* dans le temps situé entre le début et la fin de l'événement tel que l'histoire séquentielle obtenue en projetant chaque événement sur son point de linéarisation est correcte par rapport à la spécification séquentielle de T. La différence avec la cohérence séquentielle est le temps réel : dans la linéarisabilité, si le retour d'un événement e précède temporellement l'appel de l'opération d'un événement e', alors e doit être placé avant e' dans la linéarisation. Par exemple, l'histoire de la figure 2.9a n'est pas linéarisable car l'écriture du 2 précède la lecture $(0, 1)$ dans le temps mais pas dans la linéarisation. La linéarisabilité peut également être comprise en s'imaginant une copie partagée gérée par un serveur, comme sur la figure 2.9b. La différence avec la cohérence séquentielle est que, pour implémenter la linéarisabilité, les processus envoient un message au serveur et attendent sa réponse pour toutes les opérations et pas seulement pour les lectures.

Contrairement à la cohérence séquentielle, la linéarisabilité est locale (composable et décomposable). Pour cette raison, elle est le critère de cohérence le plus étudié.

Notre modélisation des histoires concurrentes ne prend pas en compte le temps réel. Il n'est donc pas possible de transcrire la linéarisabilité directement. En fait, il est possible de prendre en compte le temps réel dès la modélisation du système sous forme d'histoires concurrentes, comme le montre la figure 2.15 : les événements sont les appels aux opérations et un événement e précède un événement e' dans l'ordre de processus si la fin de e précède temporellement le début de e'. Dans ce modèle, les chaînes maximales selon l'ordre de processus ne correspondent pas à l'ensemble des

[4]Crédits : Jason Cartwright. https://www.flickr.com/photos/43917222@N00/16226740

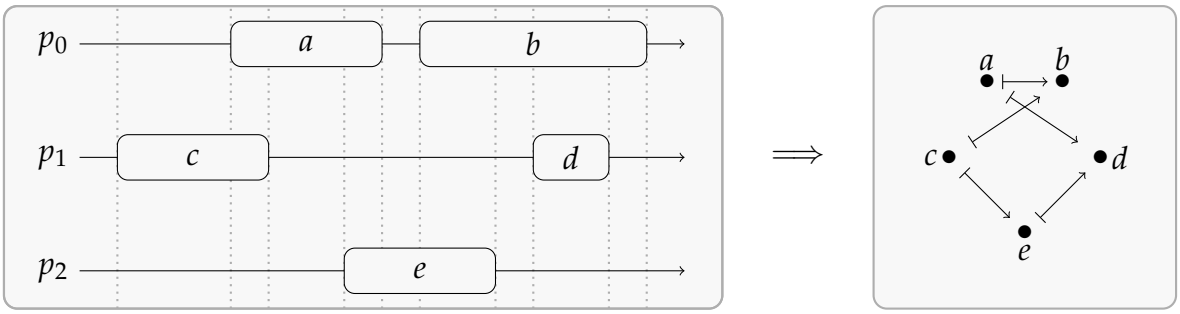


FIGURE 2.15 – Modélisation du temps réel dans les histoires concurrentes. Deux événements sont ordonnés selon l'ordre de processus si la fin du premier précède le début du second.

événements d'un processus. Par contre, l'ordre de processus est un ordre d'intervalle, ce qui signifie, entre autres, que pour tout quadruplet d'événements (e_1, e_2, e_3, e_4), si $e_1 \mapsto e_2$ et $e_3 \mapsto e_4$ alors $e_1 \mapsto e_4$ ou $e_2 \mapsto e_3$. La restriction de la cohérence séquentielle aux histoires concurrentes qui vérifient cette propriété est composable.

Calculabilité de la cohérence forte

La mémoire. L'une des questions centrales de l'algorithmique du réparti peut être formulée ainsi : « quelles sont les hypothèses minimales nécessaires pour implémenter un objet partagé ? ». Attiya, Bar-Noy et Dolev ont proposé une implémentation du registre atomique [12] dans $AS_n\left[t < \frac{n}{2}\right]$, le système réparti asynchrone de processus communiquant par messages où strictement moins de la moitié des processus peuvent tomber en panne (voir section 2.1.2).

Attiya et Welch ont montré que cette hypothèse était nécessaire [14]. Plus précisément, cet article compare le coût d'implémentation de la cohérence séquentielle et de la linéarisabilité. Les deux auteurs montrent que pour implémenter une mémoire séquentiellement cohérente dans un système synchrone dans lequel il existe une borne δ sur le temps de transmission des messages, il est nécessaire d'attendre une durée δ pour les lectures ou pour les écritures. Pour implémenter une mémoire linéarisable, l'attente est nécessaire pour les deux types d'opération. La mémoire fortement cohérente ne peut donc pas être implémentée dans les systèmes sans-attente. Remarquons que l'intuition avec un serveur que nous avons donnée pour expliquer la cohérence séquentielle et la linéarisabilité (figure 2.9) est compatible avec ce résultat : pour la cohérence séquentielle, on attendait une réponse du serveur pendant les lectures mais pas pendant les écritures, alors que l'attente était nécessaire pour toutes les opérations dans la linéarisabilité. Le même résultat est démontré pour la file.

Ce résultat est très proche du théorème CAP, initialement présenté comme une conjecture par Brewer [26] puis formalisée et prouvée par Gilbert et Lynch [48] : il est impossible d'implémenter la cohérence forte (C pour « strong consistency ») en garantissant que toutes les opérations terminent (A pour « availability ») dans un système sujet au partitionnement (P pour « partition tolerance »). Or dans des réseaux à grande échelle, des événements comme des partitions se produisent régulièrement en pratique [41, 122]. Ce point est toutefois sujet à critique [68] car la notion de système partitionable n'a jamais été définie précisément et peut prêter à confusion. La modélisation par des pertes de messages proposée par Gilbert et Lynch ne correspond pas exactement à la même notion. Dans le chapitre 5, nous prouvons des résultats plus forts et plus précis que le théorème CAP en utilisant les systèmes sans-attente pour modéliser les systèmes partitionables.

Le Consensus. Le *Consensus* est un problème fondamental en algorithmique du réparti. Pour utiliser le Consensus, chaque processus p_i *propose* une valeur v_i en utilisant l'opération propose(v_i) qui vérifie les trois propriétés suivantes.

Terminaison. Si p_i est correct, il finira par *décider* d'une valeur.

Validité. Toute valeur décidée par un processus a été proposée.

Accord. Il n'existe pas deux processus qui décident des valeurs différentes.

Le Consensus est universel : il est possible d'implémenter la machine à état, un modèle universel encodant tous les objets linéarisables proposé par Lamport [73], en utilisant l'opération propose(v_i) comme seule primitive de communication [107]. Dans [43], Fischer, Lynch et Paterson ont montré qu'il était impossible d'implémenter le Consensus dans un système asynchrone où au moins une panne franche peut survenir. Dans [37], Dolev, Dwork et Stockmayer étendent ce résultat à de nombreux systèmes.

Dans [56], Herlihy montre qu'il n'est pas possible d'implémenter le Consensus dans un système réparti composé de $n+1$ processus en utilisant un algorithme qui l'implémente dans un système composé de n processus, et ce quel que soit le nombre d'instances disponibles de l'algorithme. Il en déduit une hiérarchie pour classer les objets partagés : le *nombre de Consensus* d'un objet partagé est le nombre maximal n tel qu'il est possible d'implémenter le Consensus entre n processus en utilisant les opérations de l'objet comme seules primitives de communication. Un objet qui peut résoudre le Consensus quel que soit le nombre de processus a un nombre de Consensus infini. Tout objet partagé a un nombre de Consensus au moins égal à 1 car le Consensus avec un seul processus est trivial. Réciproquement, aucun objet qui a un nombre de Consensus supérieur ou égal à 2 ne peut être implémenté dans les systèmes où le Consensus est impossible. Le nombre de Consensus de la mémoire linéarisable est 1, celui de l'ADT file linéarisable est de 2, alors que celui de l'UQ-ADT file linéarisable est infini. Pour tout $n \geq 1$, la mémoire dans laquelle il est possible d'écrire dans n registres atomiquement a un nombre de Consensus de $2n-2$.

La hiérarchie de Herlihy est parfois critiquée pour son manque de robustesse vis-à-vis de la composition : Dans [114], Shenk montre qu'il existe des objets qui ont un nombre de Consensus strictement inférieur à n, mais dont le nombre de Consensus de la composition est n. Ce résultat est contre-intuitif car la hiérarchie est justement basée sur le fait qu'il est impossible d'implémenter des objets avec un nombre de Consensus $n+1$ en utilisant autant d'instances que nécessaire du Consensus entre n processus.

Autres critères forts

De nombreux autres critères ont été proposés comme des adaptations de la linéarisabilité ou de la cohérence séquentielle. Comme ils ne peuvent pas non plus être implémentés dans les systèmes sans-attente, nous nous contentons de les citer. La *cohérence au repos* (« quiescent consistency ») [36] est un critère intermédiaire entre la cohérence séquentielle et la linéarisabilité, dans lequel le temps réel ne doit être respecté entre deux événements que s'ils sont séparés par un temps de repos durant lequel aucune opération n'est effectuée. La k-atomicité [7], définie uniquement pour la mémoire, est un affaiblissement de la linéarisabilité qui autorise une lecture à retourner une valeur parmi les k dernières valeurs écrites, selon l'ordre total des points de linéarisation ; la linéarisabilité est donc égale à la 1-atomicité. La *linéarisabilité inéluctable* (« eventual

linearizability ») [39, 108] affaiblit la linéarisabilité en acceptant que la linéarisabilité ne soit assurée qu'à partir d'un instant t inconnu à l'avance. Enfin, la linéarisabilité d'ensembles (« set-linearisability ») [86] et la linéarisabilité d'intervalles (« interval-linearisability ») [31] étendent la linéarisabilité à des structures de données qui n'ont pas de spécifications séquentielles dans le but d'unifier la spécification des objets persistants comme les types de données abstraits que nous considérons dans cette thèse et les objets de décision à usage unique comme le Consensus. Pour cela, les auteurs proposent de modéliser les objets partagés par un modèle très proche des automates d'ordres [51] qui autorise un ensemble d'opérations à être concurrentes.

2.2.2 Les systèmes transactionnels

Les algorithmes génériques pour implémenter la cohérence forte sont très coûteux, et les algorithmes spécifiques à un objet particulier sont souvent délicats à concevoir et prouver corrects. Les transactions ont émergé parallèlement dans deux domaines différents : les systèmes de gestion de bases de données [54] et l'algorithmique du réparti en mémoire partagée [57]. Dans les deux cas, les systèmes offrent classiquement un ensemble d'opérations de base, l'insertion, la suppression et l'édition de données dans le cas des bases de données et la lecture et l'écriture dans le cas de la mémoire partagée. L'approche des *systèmes transactionnels* est de permettre d'encapsuler une suite d'opérations de base au sein de *transactions*. La cohérence est alors à la fois assurée au niveau des opérations de base et au niveau des transactions prises comme un tout.

Dans les bases de données, les propriétés attendues des transactions sont traditionnellement désignées par l'acronyme ACID [54], pour *Atomicité* — une transaction ne peut être qu'entièrement *acceptée* (« committed ») ou complètement *avortée* (« aborted ») —, *Cohérence* — les transactions exécutées dans un état correct mènent le système dans un état correct —, *Isolation* — les transactions n'interfèrent pas entre elles — et *Durabilité* — une transaction acceptée n'est pas remise en cause. Plusieurs critères de cohérence ont été proposés pour unifier la spécification des transactions dans les systèmes de mémoire transactionnelle logicielle et dans les bases de données.

La sérialisabilité

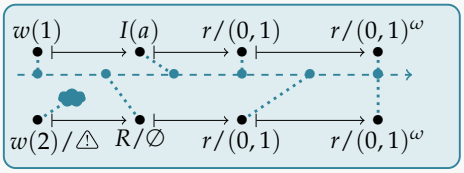
La sérialisabilité [88] est la propriété la plus souvent requise pour les systèmes de transactions. Elle impose que l'histoire formée uniquement des transactions acceptées soit constituée des mêmes événements qu'une histoire séquentielle correcte, dans laquelle les événements d'un processus se produisent dans le même ordre que sur ce processus. Autrement dit, l'histoire concurrente dans laquelle on a enlevé les transactions avortées doit être séquentiellement cohérente.

Une façon d'implémenter un objet partagé dans un système de transactions est d'encapsuler toutes les opérations de l'objet à l'intérieur d'une transaction. Dans cette thèse, ce que nous définissons comme la *sérialisabilité* (*Ser*) (figure 2.16) est le critère de cohérence que l'on obtient pour les objets implémentés de cette manière. La sérialisabilité est très proche de la cohérence séquentielle, avec la seule différence que des événements peuvent *avorter*. Un événement avorté ne fait pas partie de la linéarisation finale. En contrepartie, le processus qui a appelé l'opération est averti de l'avortement. La sérialisabilité est donc plus faible que la cohérence séquentielle, qui peut être vue comme un type de sérialisabilité dans laquelle aucun événement ne peut avorter. Pour les besoins de notre étude, nous renforçons la sérialisabilité en imposant que les lectures pures ne peuvent pas avorter.

SÉRIALISABILITÉ

✗ Non composable p. 41
✓ Décomposable p. 41
✓ Faible p. 126

La sérialisabilité [19] est un critère très proche de la cohérence séquentielle (voir page 46), à la différence que certains événements sont autorisés à avorter. *Les événements avortés ne sont pas pris en compte dans la linéarisation finale. Par contre, le processus appelant obtient l'information que son appel a échoué, car il retourne une valeur particulière ⚠. On renforce ce critère en interdisant aux lectures pures d'avorter.*
L'histoire de la figure 2.8c (rappelée ci-contre) est sérialisable : si l'on enlève l'écriture $w(2)$ qui a avorté, le reste de l'histoire est séquentiellement cohérente.



Formellement, une histoire H est sérialisable pour un type de données abstrait T (définition 2.22) s'il existe un ensemble d'événements $C \subset E_H$ (C pour « commit », du nom de l'opération appelée à la fin des transactions dans les systèmes transactionnels) tel que :

- *aucune lecture pure n'est avortée, c'est-à-dire $\hat{Q}_{T,H} \subset C$;*
- *l'histoire $H[C/C]$ composée des événements non avortés est séquentiellement cohérente ;*
- *tout événement avorté $e \in E_H \setminus C$ retourne le symbole ⚠, c'est-à-dire que son étiquette $\Lambda(e)$ est un élément de $\mathsf{A} \times \{⚠\}$ (A est l'ensemble des symboles d'entrée de T).*

Dans l'exemple ci-dessus, C contient tous les événements sauf celui étiqueté $w(2)/⚠$ et la linéarisation suivante de $H[C/C]$ respecte la spécification séquentielle :

$$w(1)/\bot \cdot R/\emptyset \cdot I(a)/\bot \cdot r/(0,1)^\omega$$

Définition 2.22 (Sérialisabilité). La *sérialisabilité* est le critère de cohérence :

$$Ser : \left\{ \begin{array}{lll} \mathcal{T} & \to & \mathcal{P}(\mathcal{H}) \\ T & \mapsto & \left\{ H \in \mathcal{H} : \begin{array}{rl} \exists C \subset E_H, & \hat{Q}_{T,H} \subset C \\ \wedge & \mathrm{lin}(H[C/C]) \cap L(T) \neq \emptyset \\ \wedge & \forall e \in E_H \setminus C, \Lambda(e) \in \mathsf{A} \times \{⚠\} \end{array} \right\} \end{array} \right.$$

Hormis l'histoire séquentiellement cohérente de la figure 2.8b, aucune autre des histoires de la figure 2.8 n'est sérialisable puisque aucun de leurs événements n'avorte et qu'elles ne sont pas séquentiellement cohérentes. C'est par exemple le cas avec l'histoire de la figure 2.8d (à droite), dans laquelle un cycle est formé par l'obligation d'insérer le a avant de le lire et l'obligation de lire le 1 après qu'il ait été écrit. Par contre, si les deux écritures du premier processus avaient été faites dans l'ordre inverse, l'histoire aurait été sérialisable.

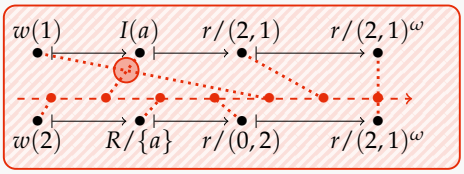


FIGURE 2.16 – La sérialisabilité.

Remarquons que cette définition de la sérialisabilité est plus large que le seul contexte des systèmes transactionnels : ce critère de cohérence est adapté pour modéliser le comportement observé pour Hangouts dans l'expérience de l'introduction (figure 1.1). Le messages de Bob n'a pas pu être envoyé, mais l'émetteur a été averti de l'échec. Cette définition de la sérialisabilité modélise donc de nombreux services client-serveur sur Internet où un message d'erreur peut être obtenu quand la connexion n'est pas assurée. L'ordre total sur les autres événements est assuré par le serveur, comme montré sur la figure 2.9c. La sérialisabilité est également très proche de la notion d'*objets avortables* (« abortable objects ») [4] dont la possibilité pour une opération d'avorter est encodée directement dans le type de données abstrait plutôt que dans un critère de cohérence. Notons toutefois que le choix de l'avortement ou non d'une opération est non-déterministe. L'approche des objets avortables n'est donc pas possible sans adaptation dans notre modèle qui ne permet pas d'encoder le non-déterminisme.

Autres critères

Les autres critères de cohérence proposés pour spécifier les systèmes transactionnels sont des variations autour de la sérialisabilité. La *sérialisabilité stricte* [88] est à la sérialisabilité ce que la linéarisabilité est à la cohérence séquentielle : elle ajoute à la sérialisabilité l'exigence que si une transaction T_1 termine avant qu'une transaction T_2 ne commence, alors T_1 sera placée avant T_2 dans l'exécution séquentielle équivalente.

Le but de la *sérialisabilité à copie unique* [18, 64], l'*atomicité globale* [123], la *récupérabilité* [52], l'*ordonnancement rigoureux* [25], la *cohérence des mondes virtuels* [60] et l'*opacité* [50] est de spécifier conjointement les transactions et les opérations de base offertes par le système transactionnel (les lectures et écritures sur la mémoire et les insertions et suppressions sur les bases de données). Ces considérations sont plutôt de l'ordre de la sémantique du langage de spécification des transactions : elles donnent des informations sur la suite d'opérations encapsulées à l'intérieur des transactions, mais offrent la même cohérence que la sérialisabilité en ce qui concerne les objets implémentés à l'aide de transactions, point de vue qui nous intéresse dans cette thèse.

À très grande échelle, garantir les propriétés ACID est très coûteux. Les entrepôts de données NoSQL (pour « not only SQL ») réduisent la cohérence qu'ils offrent au profit de la performance. Plusieurs travaux récents [29, 69, 127] proposent de garantir un critère plus faible pour gagner en efficacité : la convergence.

2.2.3 La convergence

Convergence

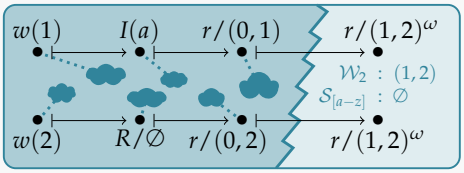
Introduite dans les années 1990 [95,119] et popularisée par Vogels dans [122], la convergence (*EC*, pour « eventual consistency ») s'applique surtout aux objets répliqués [105], pour lesquels chaque processus maintient localement une copie locale de l'objet, appelée *réplica*. La convergence impose que si tous les processus s'arrêtent d'écrire, en attendant assez longtemps, tous les réplicas finiront dans le même état : l'état de convergence. Le temps nécessaire pour atteindre la convergence après que les écritures ont cessé n'est pas spécifié et les processus ne sont pas avertis quand ils atteignent cet état. La convergence est formellement définie dans notre modèle sur la figure 2.17.

Les objets répliqués convergents actuellement déployés à grande échelle ont principalement deux applications : les entrepôts de données clé-valeur comme Dynamo [35] et Cassandra [70] d'une part et d'autre part les éditeurs collaboratifs comme Woot [87],

CONVERGENCE

✓ Composable p. 41
✗ Non décomposable p. 41
✓ Faible p. 126

La convergence est l'un des rares critères a ne pas avoir été initialement défini pour la mémoire [122]. Il impose que si tous les participants arrêtent d'écrire, tous les réplicas finissent par converger vers un état commun. Par exemple, dans l'histoire de la figure 2.8e (rappelée ci-contre), au bout d'un moment, toutes les lectures retournent la même valeur.



Formellement, une histoire H est convergente pour un type de données abstrait T (définition 2.23) si elle rentre dans l'un des deux cas suivants.

- *H contient une infinité d'écritures (c'est-à-dire $|U_{T,H}| = \infty$). Dans ce cas, au moins un participant écrit perpétuellement, donc les valeurs retournées par les lectures peuvent continuer à évoluer en fonction des écritures.*
- *Il existe un état ζ dans T (l'état de convergence) qui peut expliquer toutes les lectures faites à partir d'un certain moment (l'instant de convergence). Ce qui caractérise un tel état est le fait que l'ensemble des événements se produisant après convergence E' est cofini ($|E_H \setminus E'| < \infty$) et pour tout événement $e \in E'$, l'opération $\Lambda(e)$ peut être faite dans l'état ζ, donc $\zeta \in \delta_T^{-1}(\Lambda(e))$.*

Dans l'exemple ci-dessus, E' est l'ensemble des événements étiquetés $r/(1,2)$ et l'un des états de convergence possible est $\zeta = ((1,2),\emptyset)$. En fait, n'importe quelle valeur pour l'ensemble donne un état de convergence possible.

Définition 2.23 (Convergence). La *convergence* est le critère de cohérence :

$$EC : \left\{ \begin{array}{lll} \mathcal{T} & \to & \mathcal{P}(\mathcal{H}) \\ T & \mapsto & \left\{ H \in \mathcal{H} : \begin{array}{rll} & |U_{T,H}| = \infty & \\ \vee & \exists E' \subset E_H, & |E_H \setminus E'| < \infty \\ & \wedge & \bigcap_{e \in E'} \delta_T^{-1}(\Lambda(e)) \neq \emptyset \end{array} \right\} \end{array} \right.$$

L'histoire de la figure 2.8g (rappelée ci-contre) n'est pas convergente puisqu'elle ne contient que trois écritures (étiquetées $w(1)/\perp$, $w(2)/\perp$ et $I(a)/\perp$) mais, pour une infinité de lectures, le flux fenêtré doit être dans l'état $(1,2)$ et pour une autre infinité de lectures, l'état doit être $(2,1)$.

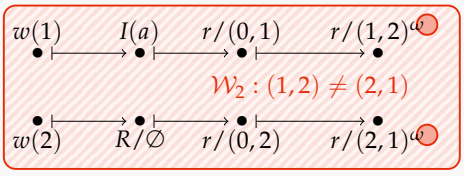


FIGURE 2.17 – La convergence.

Logoot [124], TreeDoc [97] et LSEQ [85]. Remarquons que la convergence est le critère de cohérence qui modélise le comportement observé pour Skype dans l'expérience de l'introduction (figure 1.3) : les deux interlocuteurs n'ont pas reçu les messages dans le même sens, mais l'ordre d'affichage a été réordonné pour qu'ils les voient finalement dans le même ordre.

La convergence est composable mais pas décomposable. Sur la figure 2.18 trois processus partagent une mémoire composée de deux registres. Seuls les événements répétés une infinité de fois importent ici : des écritures sur le registre x et des lectures contradictoires de 1 et 2 sur y. Cette histoire est convergente uniquement parce qu'elle contient une infinité d'écritures sur x, mais si l'on ne considère que l'histoire

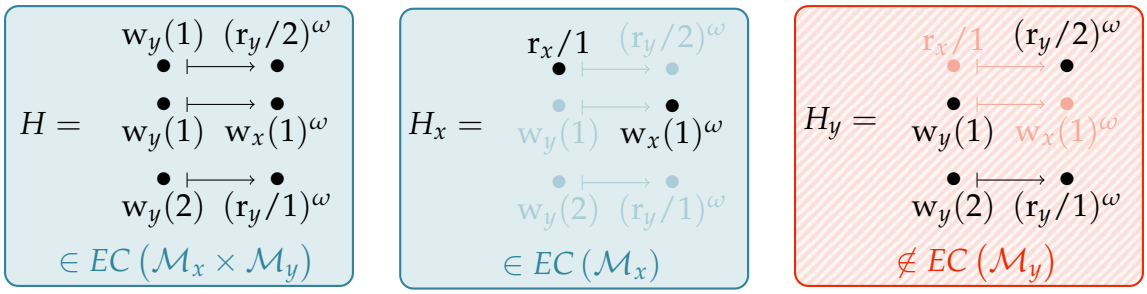


FIGURE 2.18 – La convergence n'est pas décomposable. L'objet partagé de l'histoire H est une instance de mémoire $\mathcal{M}_{\{x,y\}} = \mathcal{M}_x \times \mathcal{M}_y$ et les histoires H_x et H_y sont projections de H qui ne conservent que les événements concernant les registres $\mathcal{M}_x$ et $\mathcal{M}_y$ respectivement. L'histoire H est convergente parce que les écritures ne cessent pas, mais pas l'histoire H_y.

qui concerne le registre y, la convergence n'est jamais atteinte. De manière générale, au moins l'un des objets d'une composition est convergent, mais tous le ne sont pas forcément.

De plus, en raison de la restriction qui empêche les lectures pures d'avorter dans la sérialisabilité, celle-ci est également plus forte que la convergence. En effet, dans une histoire possédant un nombre fini d'écritures et une infinité de lectures, presque toutes celles-ci seront faites dans l'état obtenu par la linéarisation imposée par la sérialisabilité sur les écritures qui n'ont pas avorté.

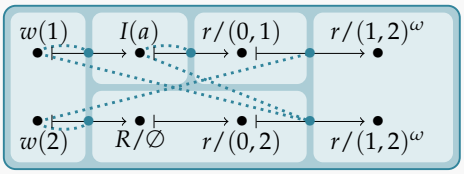
Implémentation et convergence forte

Pour atteindre la convergence, les conflits peuvent être résolus soit à l'écriture, soit à la lecture, soit de façon asynchrone. La convergence forte (*SEC*, pour « Strong Eventual Consistency ») [112] impose la résolution à l'écriture : si deux réplicas ont reçu les mêmes messages, ils doivent être dans le même état. Le problème de cette définition est que les notions de réplica et de réception de messages sont inhérentes à l'implémentation, et ne sont pas accessibles au programmeur qui utilise l'objet. Ils ne devraient donc pas apparaître tels quels dans la spécification. Une relation de visibilité est introduite [28] pour modéliser la notion de réception de message. La visibilité n'est pas une relation d'ordre car elle n'est pas nécessairement transitive. Comme son nom l'indique, la convergence forte, définie sur la figure 2.19, est plus forte que la convergence.

Les *types de données répliqués commutatifs* (*CmRDT* pour « Commutative Replicated Data Types ») [111] sont des types de données abstraits dont toutes les écritures commutent deux à deux. Par exemple, un compteur que l'on peut incrémenter ou décrémenter est un CmRDT. L'état obtenu étant indépendant de l'ordre d'exécution des opérations, il est facile d'implémenter la convergence forte pour les CmRDT. De façon similaire, la convergence forte est facile à obtenir pour les *types de données répliqués convergents* (*CvRDT* pour « Convergent Replicated Data Types »), types de données abstraits dont l'ensemble des états forme un semi-treillis et dont toutes les opérations sont croissantes. La structure qui permet de proposer une valeur entière et qui converge vers la plus grande valeur proposée est un exemple de *CvRDT*. Dans [111], Shapiro, Preguiça, Baquero et Zawirski prouvent que ces deux classes sont égales. Le terme *type de données répliqué sans conflit* (*CRDT*, pour « Conflict-free Replicated Data Types ») représente cette classe commune.

CONVERGENCE FORTE

✓ Composable p. 41
✗ Non décomposable p. 41
✓ Faible p. 126



La convergence forte renforce la convergence en imposant aux processus de converger dès qu'ils ont reçu les messages correspondant aux mêmes écritures. Dans l'histoire de la figure 2.8e (rappelée ci-contre), les opérations faites par des processus qui ont reçu les mêmes messages sont représentées dans le même cadre clair.

Formellement, une histoire H est fortement convergente pour un type de données abstrait T (définition 2.25) s'il existe une relation de visibilité $\xrightarrow{\text{VIS}}$ (qui modélise la réception des messages, définition 2.24) tel que l'état lu lors d'un événement e est entièrement déterminé par l'ensemble $U_{T,H} \cap \lfloor e \rfloor_{\xrightarrow{\text{VIS}}} \setminus \{e\}$ des écritures visibles par e, et une fonction f qui décrit dans quel état les lectures doivent être faites en fonction de ses écritures visibles. Rappelons que $\delta_T^{-1}(\Lambda(e))$ est l'ensemble des états dans lesquels un événement e peut avoir lieu (définition 2.3).
Dans l'exemple ci-dessus, une relation de visibilité possible est dessinée en pointillés. L'une des façons de définir la fonction f est de dire que l'ensemble reste toujours vide et que l'état du flux fenêtré est la séquence des k plus petites valeurs écrites triées par ordre croissant sur les entiers. Aucun lien n'est en effet imposé entre la spécification séquentielle et la fonction f.

Définition 2.24 (Relation de visibilité)**.** Soit $H \in \mathcal{H}$ une histoire concurrente. Une relation binaire $\xrightarrow{\text{VIS}} \in E_H^2$ est une *relation de visibilité* de H si :

- elle contient l'ordre de processus $\mapsto$ de H ;
- elle est réflexive et acyclique ;
- toute opération finit par devenir visible par tous les processus, c'est-à-dire

$$\forall e \in E_H, \{e' \in E, e \xrightarrow{\text{VIS}} e'\} \text{ est fini;}$$

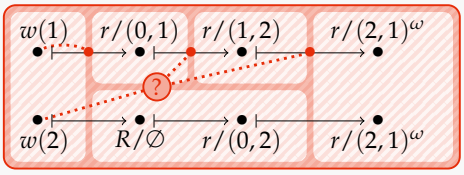
- une opération visible par un processus reste toujours visible par ce même processus

$$\forall e, e', e'' \in E_H, \left(e \xrightarrow{\text{VIS}} e' \wedge e' \mapsto e''\right) \Rightarrow \left(e \xrightarrow{\text{VIS}} e''\right).$$

L'ensemble des relations de visibilité de H est noté $Vis(H)$.

Définition 2.25 (Convergence forte)**.** La *convergence forte* est le critère de cohérence

$$SEC : \left\{ \begin{array}{ccc} \mathcal{T} & \to & \\ T & \mapsto & \left\{ H \in \mathcal{H} : \begin{array}{c} \mathcal{P}(\mathcal{H}) \\ \exists \xrightarrow{\text{VIS}} \in Vis(H), \exists f : \mathcal{P}(U_{T,H}) \to Z, \\ \forall e \in E_H, f\left(U_{T,H} \cap \lfloor e \rfloor_{\xrightarrow{\text{VIS}}} \setminus \{e\}\right) \in \delta_T^{-1}(\Lambda(e)) \end{array} \right\} \end{array} \right.$$



L'histoire de la figure 2.8f (rappelée ci-contre) n'est pas fortement convergente. En effet, le premier processus change deux fois d'état après l'écriture $w(1)$ mais l'histoire ne comporte qu'une seule autre écriture. Si la lecture $r/(1,2)$ a l'écriture $w(2)$ dans son passé visible, elle a le même passé visible que les lectures $r/(2,1)$. Sinon, elle a le même passé visible que la lecture $r/(0,1)$. Les deux cas sont en contradiction avec la définition de la convergence forte.

FIGURE 2.19 – La convergence forte.

Le même principe est utilisé dans le langage de programmation Bloom [9], qui dispose d'un analyseur syntaxique, CALM [8], pour s'assurer que les opérations sont bien commutatives.

De nombreux exemples de CRDT peuvent être trouvés dans [112]. Beaucoup d'autres types de données utiles en pratique ne sont en revanche pas des CRDT. Par exemple, l'insertion d'un élément dans un ensemble ne commute pas avec la suppression du même élément. La notion de CRDT est étendue à certains de ces objets. L'ensemble partagé est l'une des structures de données les plus étudiées du point de vue de la convergence. Parmi toutes les implémentations de l'ensemble proposées à partir de CRDT, on peut citer le *G-Set* (pour « Grow-Only Set ») [112] qui n'autorise que l'insertion d'éléments ; le *2P-Set* (pour « two-phases set », aussi appelé *U-Set*) [126] qui sépare les éléments insérés et les éléments supprimés en deux G-Set représentant une liste blanche et une liste noire et calcule la différence ensembliste lors des lectures ; l'OR-Set (Observed-Remove Set) [21,84,112] construit sur le même principe mais dans lequel chaque élément est associé à un identifiant unique, ce qui permet de réinsérer un élément après sa suppression ; le C-Set (« Counter Set ») [11] et le LWW-Set (« Last-Writer-Wins Set ») [112] qui associent, à chaque élément, des compteurs dont la valeur déterminent s'ils appartiennent à l'ensemble ou non.

Toutes ces variations autour de l'ensemble partagé ont des comportements très différents. On peut donc se poser la question de leur correction. Quel est leur lien avec le type de données abstrait de l'ensemble ? Dans [21], une critique est émise envers le C-Set : si tous les processus insèrent la même valeur lors de leur dernière écriture, l'élément peut tout de même être absent après convergence. L'OR-Set, qui assure que l'élément sera présent dans la même situation, serait donc plus conforme à l'intuition que le C-Set. Pourtant, l'OR-Set présente des incohérences semblables (voir chapitre 3). Cela pose la question de la spécification de ces objets.

Spécifier les objets convergents

Si la convergence et la convergence forte ne sont pas suffisantes pour spécifier complètement les objets partagés, c'est parce qu'elles ne disent rien sur l'état dans lequel les réplicas doivent converger. N'importe quel état de convergence est donc *a priori* correct. Trois approches ont principalement été étudiées pour spécifier l'état de convergence.

L'intention. La notion d'*intention* [75,117] a initialement été proposée pour spécifier les éditeurs collaboratifs convergents. L'idée est de prendre en compte l'intention de l'utilisateur qui a appelé les opérations dans la résolution de l'état de convergence. Par exemple, si le document partagé contient le mot « *balade* » et que les deux opérations $I(l,3)$ (insertion d'un l en troisième position) et $I(s,7)$ sont appelées simultanément, la deuxième opération doit être transformée en $I(s,8)$ pour obtenir l'état de convergence « *ballades* » car l'intention de l'utilisateur était de mettre le mot au pluriel. Cependant, la notion d'intention n'a, à notre connaissance, jamais pu être formellement définie et n'a pas eu d'impact en dehors de l'édition collaborative.

Le principe de l'équivalence des permutations. Le principe de l'équivalence des permutations a été proposé dans [21]. Il définit la conformité d'un objet à sa spécification séquentielle par : « si toutes les permutations séquentielles des opérations mènent à un état équivalent, alors l'exécution concurrente de ces opérations devrait mener à un état équivalent ». Pour spécifier complètement un objet convergent, il suffirait alors d'attribuer un état de convergence à chaque paire d'opérations non-commutatives.

Par exemple, dans le cas de l'ensemble partagé, deux insertions commutent. Il en va de même pour deux suppressions, ainsi que pour l'insertion et la suppression de deux éléments différents. Pour spécifier complètement l'ensemble partagé, il suffirait donc de définir le comportement de chaque opération et de décider ce que doit donner l'exécution concurrente de l'insertion et de la suppression du même élément. Dans le cas de l'OR-Set, l'état obtenu est celui qui contient l'élément.

Le principe des permutations équivalentes semble acceptable pour définir l'ensemble partagé car pour chaque opération, il existe au plus une opération avec laquelle elle ne commute pas. Qu'en est-il si ce n'est pas le cas ? Prenons l'exemple d'un type de données abstrait possédant trois états ζ_a, ζ_b et ζ_c, et trois opérations a, b et c. Exécutée seule, a mène dans l'état ζ_a, b dans l'état ζ_b et c dans l'état ζ_c. Aucun couple d'opérations différentes ne commute. Selon le principe des permutations équivalentes, il suffirait de définir un comportement pour chaque couple. Une possibilité est de définir que $a||b$ mène dans l'état ζ_c, $b||c$ mène dans l'état ζ_a et $a||c$ mène dans l'état ζ_b. Mais alors que doit-on obtenir si l'on exécute a, b et c en parallèle ? Les trois états jouent un rôle symétrique donc aucun n'est plus légitime pour être choisi comme état de convergence.

Un exemple plus concret est le C-set. Pour les exemples simples avec uniquement deux opérations, l'exécution est conforme aux attentes. Le contre-exemple qui a été donné dans [21] met en jeu quatre opérations.

Spécifications concurrentes. En pratique, expliciter le comportement des opérations deux à deux n'est pas suffisant pour spécifier complètement un objet. Dans le cas général, une opération n'est pas seulement concurrente à une autre opération, mais à un ensemble d'opérations, elles-mêmes reliées entre elles selon la relation de visibilité de la convergence forte. Burckhardt, Gotsman, Yang et Zawirski [28] proposent la notion de *spécification concurrente* pour spécifier les objets partagés fortement convergents.

Formellement, l'approche spécifie l'état retourné lors d'une opération par le *contexte d'opération* dans lequel elle est effectuée (définition 2.26), c'est-à-dire l'ensemble des opérations de son passé selon la relation de visibilité, ordonnées selon la relation de visibilité et une relation d'*arbitrage*, ordre total pouvant être utilisée pour résoudre les conflits. Une spécification concurrente associe une valeur de retour à chaque opération en fonction du contexte dans lequel elle est appelée (définition 2.27).

Définition 2.26 (Contexte d'opération)**.** Soit A un ensemble dénombrable, qui joue le même rôle que l'alphabet d'entrée dans les types de données abstraits. Un *contexte d'opération* sur A est un quadruplet $L = (E, \Lambda, \xrightarrow{\text{VIS}}, \leq)$ où :

- E est un ensemble fini d'*événements*, comme dans les histoires concurrentes ;
- $\Lambda : E \to \Sigma$ est la *fonction d'étiquetage* qui joue le même rôle que dans les histoires concurrentes ;
- $\xrightarrow{\text{VIS}}$ est une *relation de visibilité* sur les événements de E (définition 2.24)
- $\leq$, l'*arbitrage*, est un ordre total sur les événements de E.

L'ensemble des contextes d'opération sur A est noté $\mathcal{L}(\mathsf{A})$

Définition 2.27 (Spécification concurrente)**.** Une *spécification concurrente* est un triplet $(\mathsf{A}, \mathsf{B}, F)$ où :

- A et B sont des ensembles dénombrables appelés *alphabets d'entrée* et de *sortie* ;
- $F : \mathcal{L}(\mathsf{A}) \times \mathsf{A} \to \mathsf{B}$ est la fonction de spécification qui indique quel symbole de retour est attendu pour chaque symbole d'entrée dans un contexte d'opération donné.

Définition 2.28 (Convergence forte pour une spécification concurrente)**.** Soient $H = (\Sigma, E, \Lambda, \mapsto, \nearrow)$ une histoire concurrente et $(\mathsf{A}, \mathsf{B}, F)$ une spécification concurrente.

On dit que H est fortement convergente par rapport à $(\mathsf{A}, \mathsf{B}, F)$ s'il existe une relation de visibilité $\xrightarrow{\text{VIS}}$ et un ordre total $\leq$ sur E_H (l'arbitrage) tels que pour tout événement $e \in E$, l'opération qui étiquette e est conforme à l'application de la spécification concurrente sur le contexte formé par le passé de e selon la relation de visibilité :

$$\forall e \in E, \Lambda(e) = \alpha/\beta \Leftrightarrow F((\lfloor E \rfloor_{\xrightarrow{\text{VIS}}}, \Lambda, \xrightarrow{\text{VIS}}, \leq), \alpha) = \beta$$

La spécification concurrente de l'OR-Set est appelée *IW-Set* (pour « Insert Wins Set »), formellement définie sur la figure 2.20. Elle exprime le fait qu'un élément x appartient à l'ensemble si et seulement si un événement d'insertion de x est visible au moment de la lecture, mais aucun événement de suppression de x n'est situé entre l'insertion et la lecture selon $\xrightarrow{\text{VIS}}$.

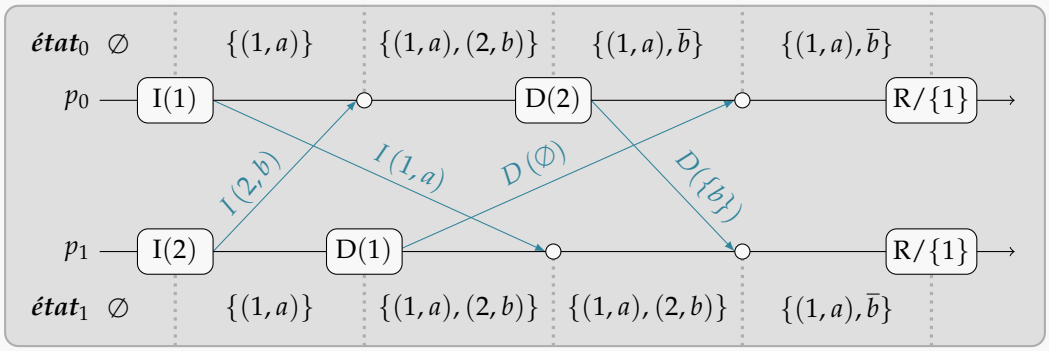
En pratique, cette approche présente également plusieurs limites.

1. La première limite est sa complexité par rapport aux spécifications séquentielles. Les types de données abstraits sont spécifiés grâce à des systèmes de transitions dans lesquelles la spécification de chaque opération est indépendante. Les systèmes de transitions décrivent naturellement un langage : la spécification séquentielle. Dans les spécifications concurrentes, au contraire, il est nécessaire de donner une spécification pour tous les contextes d'opération possibles, ce qui mène généralement à une spécification aussi complexe que l'implémentation elle-même.

   Cette limite pose deux questions. Premièrement, quel est le rapport entre les spécifications concurrentes et l'intuition que l'on se fait des structures de données qu'elles prétendent spécifier ? En bref, les spécifications concurrentes ne permettent pas de justifier pourquoi l'OR-Set serait plus conforme à l'intuition d'un ensemble que le C-Set, comme cela est affirmé dans [21]. Deuxièmement, les techniques de vérification existantes sont basées sur les systèmes de transitions. Quelles techniques pourraient être utilisées pour vérifier formellement des spécifications concurrentes ?

2. La deuxième limite est que les spécifications concurrentes ne peuvent être utilisées que pour spécifier les objets partagés vérifiant un critère de cohérence au moins aussi fort que la convergence forte. En effet, elles se reposent sur la notion de visibilité qui est l'une des caractéristiques de la convergence forte. La fonction F étant déterministe, deux états ayant le même passé selon la relation de visibilité sont forcément dans le même état. Par exemple, comment spécifier les objets convergents dont les conflits sont réglés de façon asynchrone selon cette technique ?

OR-SET



Le OR-Set (pour « Observed-Remove Set ») [112] est l'une des adaptations des CRDT à l'ensemble partagé. Il possède les mêmes opérations que le type de données abstrait ensemble (page 27) : l'insertion et la suppression d'un élément v et la lecture des éléments présents dans l'ensemble. Lors d'une insertion, une étiquette unique à l'histoire est accolée à l'élément inséré. Par exemple, dans l'exécution ci-dessus, les étiquettes a et b sont accolées aux deux insertions. Pour supprimer un élément v, le processus commence par observer *les étiquettes accolées à l'élément à supprimer (d'où le nom de la structure de données), et envoie un message contenant ces étiquettes. Par exemple, lors de la suppression du* 2*, le premier processus observe l'élément $(2, b)$ dans son réplica local et envoie donc le message demandant la suppression de l'étiquette b. L'élément est alors supprimé du réplica local, mais une* pierre tombale $\bar{b}$ *marquant l'étiquette supprimée est conservée pour indiquer qu'elle a déjà été supprimée. Les optimisations de l'OR-Set [21, 84] visent à supprimer ces pierres tombales pour réduire le coût en mémoire.*

Quand une insertion et une suppression du même élément sont concurrentes (au sens de la relation de visibilité), la suppression n'a aucun effet. Dans l'exemple ci-dessus, l'insertion du 1 *et sa suppression sont concurrentes et le* 1 *est bien présent dans l'ensemble après convergence. Cela correspond à la stratégie « l'insertion gagne » qui est formalisée par la spécification concurrente de l'OR-Set : le IW-Set (définition 2.29). Cette spécification signifie dans le formalisme de la définition 2.27 qu'un élément v est lu pendant la lecture s'il existe une insertion de v (événement e_i étiqueté $\mathrm{I}(v)/\bot$) qui précède la lecture selon la relation de visibilité mais pas de suppression de v (événement e_d étiqueté $\mathrm{D}(v)/\bot$) entre l'insertion et la lecture.*

Définition 2.29 (IW-Set)**.** Soit *Val* un ensemble dénombrable. L'IW-Set de support *Val* est la spécification concurrente :

$$\left(\mathsf{A} = \bigcup_{n \in Val} \{\mathrm{I}(n); \mathrm{D}(n); \mathrm{R}\}, \mathsf{B} = \mathcal{P}_{<\infty}(Val) \cup \{\bot\}, F\right)$$

$$F : \begin{cases} \mathcal{L}(\mathsf{A}) \times \mathsf{A} & \to \mathsf{B} \\ \left(\left(E, \Lambda, \xrightarrow{\text{VIS}}, \leq\right), \ \mathrm{I}(n)\right) & \mapsto \bot \\ \left(\left(E, \Lambda, \xrightarrow{\text{VIS}}, \leq\right), \ \mathrm{D}(n)\right) & \mapsto \bot \\ \left(\left(E, \Lambda, \xrightarrow{\text{VIS}}, \leq\right), \ \mathrm{R}\right) & \mapsto \left\{ v \in Val : \begin{array}{l} \exists e_i \in \lfloor e \rfloor_{\xrightarrow{\text{VIS}}}, \Lambda(e_i) = \mathrm{I}(v)/\bot \wedge \forall e_d \in E \\ e_i \xrightarrow{\text{VIS}} e_d \xrightarrow{\text{VIS}} e \Rightarrow \Lambda(e_d) \neq \mathrm{D}(v)/\bot \end{array} \right\} \end{cases}$$

FIGURE 2.20 – L'OR-Set.

3. La troisième limite est que la relation de visibilité est directement calquée sur le transit des messages dans le système. Intuitivement, deux événements a et b sont concurrents si le processus qui exécute b n'a pas reçu le message de celui qui exécute a au moment où il lance b, et réciproquement. Or les messages dépendent de l'implémentation, et non de la spécification.

Finalement, on peut dire que cette approche est une bonne technique pour *modéliser* certains objets partagés : elle permet de décrire finement le comportement de ces objets et de prouver certaines propriétés sur la correction et la complexité des algorithmes, comme illustré dans [28]. Elle n'est en revanche pas satisfaisante comme technique de spécification. Dans le chapitre 3, nous présenterons un nouveau critère de cohérence, la *cohérence d'écritures*, qui utilise les spécifications séquentielles pour spécifier les états de convergence.

2.2.4 La mémoire partagée

Outre les systèmes à passage de messages à grande échelle, la mémoire fortement cohérente est également trop coûteuse dans les architectures parallèles à mémoire partagée utilisées pour les calculs à haute performance. De nouveaux modèles de mémoire ont dû être imaginés pour affaiblir la linéarisabilité. Ces nouveaux modèles ne peuvent pas être décrits par quelques propriétés comme les *garanties de session* [118] et de nombreux critères faibles ont été proposés pour spécifier des types de mémoire partagée. Plusieurs études tentent de les répertorier [1, 2, 82].

La mémoire PRAM

Dans [76], Lipton et Sandberg proposent la mémoire *PRAM* (pour « Pipelined Random Access Memory », aussi appelée mémoire FIFO) qui assure que toutes les écritures sont vues par tous les processus et que l'ordre de processus est respecté. Les auteurs proposent une spécification de PRAM : une histoire H est PRAM si, pour tout processus p, il existe une linéarisation contenant toutes les écritures sur les registres et les lectures faites par p telle que chaque lecture d'un registre x retourne la dernière valeur écrite sur x dans l'ordre de la linéarisation si elle existe ou la valeur initiale 0 sinon.

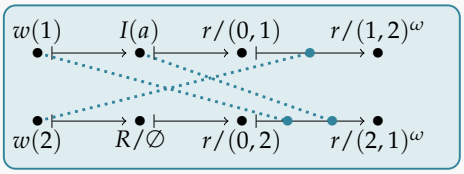
Autrement dit, la linéarisation demandée pour chaque processus vérifie la spécification séquentielle de la mémoire. Cette propriété est donc facile à étendre aux autres types de données abstraits. Nous définissons formellement le critère de cohérence correspondant, la cohérence pipeline (*PC*, pour « pipeline consistency »), sur la figure 2.21.

La cohérence pipeline est plus faible que la cohérence séquentielle, selon laquelle l'ordre de la linéarisation de tous les processus est le même. Elle n'est pas comparable avec la sérialisabilité et la convergence car exécuter des écritures non-commutatives dans un ordre différent entraîne un résultat différent.

Remarquons que la cohérence pipeline est le critère de cohérence qui modélise le comportement observé pour WhatsApp dans l'expérience de l'introduction (figure 1.2) : chaque interlocuteur reçoit tous les messages et le fait que les messages sont placés à la fin de la file de messages lors de leur réception permet d'assurer que l'évolution du contenu de l'écran de chaque interlocuteur au cours du temps est conforme à la spécification séquentielle du flux fenêtré.

COHÉRENCE PIPELINE

✗ Non composable p. 41
✓ Décomposable p. 41
✓ Faible p. 126



Selon la mémoire PRAM [76], pour chaque processus p, il existe une linéarisation qui contient toutes les lectures de p et les écritures de tous les processus, qui respecte la spécification séquentielle de la mémoire. On peut appliquer la même définition avec l'objet de la figure 2.8g (rappelée ci-contre) : l'écriture $w(2)$ du deuxième processus peut se placer entre les lectures retournant $(0,1)$ et $(1,2)$ du premier processus et les deux écritures du premier processus peuvent se placer entre les lectures $r/(0,2)$ et $r/(2,1)$ du deuxième processus.

Le niveau de formalisme adopté dans [76] pour spécifier la mémoire PRAM étant satisfaisant, nous nous contentons d'adapter la définition dans notre formalisme. Formellement, les écritures correspondent à la transition des opérations (symboles de l'alphabet d'entrée α) et les lectures à leur valeur de retour (symboles de l'alphabet de sortie β). L'opérateur de projection permet de séparer la partie écriture de la partie lecture d'une opération : si H est une histoire concurrente et p est un processus, l'histoire $H[E_H/p]$ contient les mêmes événements que H, mais les valeurs de retour (lectures) des événements qui ne sont pas dans p sont cachées. Ainsi, une histoire H vérifie la cohérence pipeline pour un type de données abstrait T (définition 2.30) si, pour chaque processus $p \in \mathcal{P}_H$, il existe une linéarisation de $H[E_H/p]$ qui appartient à $L(T)$. Dans l'histoire ci-dessus, les mots suivants sont des linéarisations pour les deux processus :

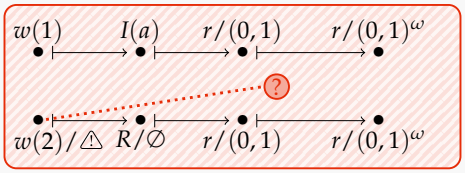
$$w(1)/\bot \cdot I(a)/\bot \cdot r/(0,1) \cdot w(2) \cdot R \cdot r \cdot (r \cdot r/(1,2))^\omega$$
$$w(2)/\bot \cdot R/\emptyset \cdot r/(0,2) \cdot w(1) \cdot I(a) \cdot r \cdot (r \cdot r/(2,1))^\omega$$

Définition 2.30 (Cohérence pipeline)**.** La *cohérence pipeline* est le critère de cohérence :

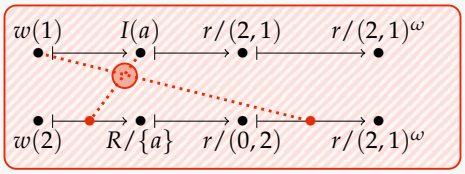
$$PC : \begin{cases} \mathcal{T} & \to & \mathcal{P}(\mathcal{H}) \\ T & \mapsto & \{H \in \mathcal{H} : \forall p \in \mathcal{P}_H, \mathrm{lin}(H[E_H/p]) \cap L(T) \neq \emptyset\} \end{cases}$$

La cohérence pipeline garantit trois propriétés.

1. Chaque processus intègre toutes les opérations d'écriture. *L'histoire de la figure 2.8c (à droite) ne respecte pas cette propriété : aucun processus ne voit l'écriture $w(2)$. Ainsi, dans n'importe quelle linéarisation pour le premier processus, l'écriture $w(2)$ est suivie d'une infinité de lectures $r/(0,1)$, ce qui ne respecte pas la spécification séquentielle.*



2. L'ordre de processus entre les écritures est respecté. *L'histoire de la figure 2.8d (à droite) ne respecte pas cette propriété car le deuxième processus voit l'écriture $I(a)$ avant la lecture $R/\{a\}$ et l'écriture $w(1)$ après la lecture $r/(0,2)$, donc il y a une inversion entre $I(a)$ et $w(1)$.*



3. L'ordre des opérations déjà intégrées n'est jamais remis en cause. *Cela est dû au fait qu'une seule linéarisation est imposée par processus : pour toute lecture, l'état retourné est conforme à l'état obtenu en appliquant toutes les écritures qui la précèdent dans l'ordre de la linéarisation. L'histoire de la figure 2.8e (à droite) ne respecte pas cette propriété car à l'intégration de l'écriture $w(1)$, le flux fenêtré du deuxième processus passe de l'état $(0,2)$ à l'état $(1,2)$, ce qui n'est pas conforme à sa spécification séquentielle.*

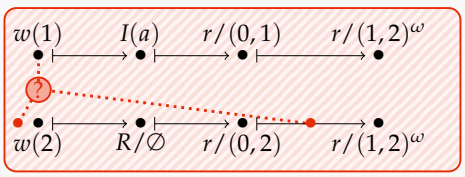
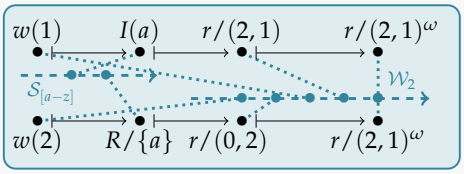


FIGURE 2.21 – La cohérence pipeline.

Cohérence de Cache

✓ Composable p. 41
✓ Décomposable p. 41
✗ Fort p. 126

La cohérence séquentielle n'est pas composable, mais que se passe-t-il si l'on utilise plusieurs objets séquentiellement cohérents dans le même programme ? Pour la mémoire, le critère de cohérence dans lequel chaque registre est séquentiellement cohérent indépendamment des autres est appelé la cohérence de cache *[115]. Plus généralement, une histoire H vérifie la cohérence de cache avec un produit de types de données abstraits $T_1 \times \cdots \times T_n$ (définition 2.31) si toutes les sous-histoires ne contenant les événements que d'un seul ADT sont séquentiellement cohérentes. Par exemple, l'histoire de la figure 2.8d (rappelée ici) vérifie la cohérence de cache : les linéarisations suivantes sont correctes pour l'ensemble et pour le flux fenêtré.*



$$I(a)/\perp \cdot R/\{a\} \qquad\qquad w(2)/\perp \cdot r/(0,2) \cdot w(1)/\perp \cdot r/(2,1)^\omega$$

Définition 2.31 (Cohérence de cache)**.** La *cohérence de cache* est la fermeture par composition $SC^\star$ de la cohérence séquentielle (voir définition 2.20).

L'histoire de la figure 2.8e (ci-contre) ne vérifie pas la cohérence de cache car la sous-histoire constituée uniquement des événements qui concernent le flux fenêtré n'est pas séquentiellement cohérente.

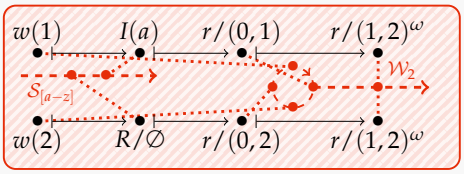


FIGURE 2.22 – La cohérence de cache.

La cohérence de cache et la mémoire lente

Dans les systèmes parallèles à grande échelle, l'accès à la mémoire partagée se fait généralement par l'intermédiaire de caches locaux à chaque processus qui permettent des écritures très rapides et garantissent une mise à jour asynchrone. À tout instant, la dernière version de chaque registre peut être dans les caches de différents processus. Si un processus lit deux registres dans son propre cache, il peut obtenir la dernière version pour l'un des registres mais une version plus ancienne pour l'autre. Le critère de cohérence assuré par un tel système est appelé *cohérence de cache* (« cache consistency » [49] ou simplement « coherence » [46]). Dans [115], Sorin, Hill et Wood montrent qu'une telle mémoire assure la cohérence séquentielle sur chaque registre séparément. La cohérence séquentielle n'étant pas composable, elle n'est pas assurée sur l'ensemble de la mémoire. Dans [3], Adve dresse la liste des conditions qu'un programme doit vérifier pour garantir une exécution séquentiellement cohérente en utilisant une mémoire vérifiant la cohérence de cache.

La caractérisation de Sorin, Hill et Wood est utile pour étendre le critère à d'autres objets que les registres. Pour la mémoire, la cohérence de cache est locale (composable et décomposable) : pour un registre unique, la cohérence de cache est équivalente à la cohérence séquentielle, donc une mémoire vérifie la cohérence de cache si, et seulement si, tous les registres qui la composent la vérifient. C'est également le plus fort critère composable plus faible que la cohérence séquentielle puisque les registres doivent vérifier la cohérence séquentielle. La cohérence de cache est donc le critère $SC^\star$ (voir figure 2.22). La cohérence de cache peut être rapprochée de la cohérence séquentielle par objet implémentée dans le système de données de Yahoo ! [34].

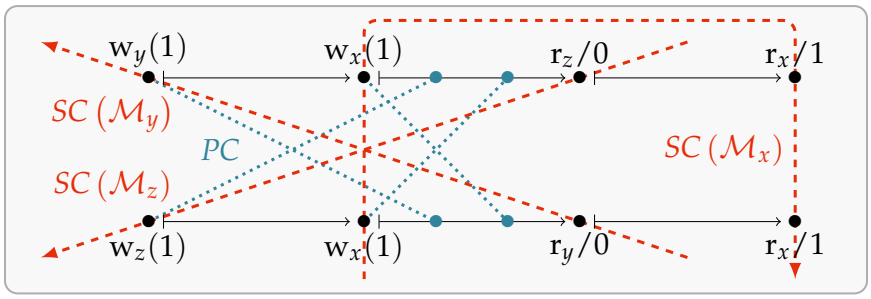


FIGURE 2.23 – La cohérence de processus est plus que la conjonction de la mémoire PRAM et de la cohérence de cache. Les deux processus partagent une instance de mémoire $\mathcal{M}_{\{x,y,z\}}$. Le premier processus écrit 1 dans y et dans x puis lit z puis x. Le deuxième processus écrit 1 dans z et dans x puis lit y puis x. Cette histoire vérifie la cohérence pipeline (pointillés bleus) et la cohérence de cache (tirets rouges) mais pas la cohérence de processus car les deux écritures sur x ne sont pas dans le même ordre dans les deux linéarisations de la cohérence pipeline.

Par définition de la fermeture par composition, aucun critère de cohérence plus faible que la cohérence séquentielle qui n'est pas plus faible que la cohérence de cache n'est composable. En particulier, comme PC n'est pas comparable à $SC^\star$, il n'existe pas de critère de cohérence composable entre la cohérence pipeline et la cohérence séquentielle.

Remarque 2.4. Pour tout critère $C \in \mathcal{C}$, si $PC \leq C \leq SC$, alors C n'est pas composable.

La mémoire lente (« slow memory ») [59] est construite de manière similaire à la cohérence de cache, mais la cohérence pipeline remplace la cohérence séquentielle. Il s'agit donc du critère de cohérence $PC^\star$.

La cohérence de processus

La *cohérence de processus* a initialement été introduite en même temps que la cohérence de cache [49], puis formellement définie dans [5]. Comme la mémoire PRAM, la cohérence de processus impose l'existence, pour chaque processus p, d'une linéarisation contenant toutes les écritures et les lectures de p. De plus, deux écritures sur le même registre doivent apparaître dans le même ordre dans les linéarisations de tous les processus. Nous ne donnons pas de définition formelle de la cohérence de processus dans notre modèle, car ce critère ne nous sera pas utile par la suite.

Ce critère est intéressant car il montre la limite de la structure de treillis pour étudier les critères de cohérence. D'après la définition, on aimerait définir la cohérence de processus comme la conjonction de la cohérence pipeline et de la cohérence de cache ($PC + SC^\star$). Dans [5], il est remarqué que la cohérence de processus est en fait strictement plus forte. La figure 2.23 illustre ce point en présentant une histoire vérifiant à la fois la cohérence pipeline et la cohérence de cache mais pas la cohérence de processus. Par la suite, nous ne chercherons pas à prouver qu'un critère C peut être décomposé en la conjonction de deux critères $C_1 + C_2$ car il existe en général quelque chose (les linéarisations de chaque processus dans le cas de la cohérence de processus) qui lie les propriétés de C_1 et de C_2 dans C.

La mémoire causale

La *mémoire causale* [6] renforce la mémoire PRAM en imposant que l'ordre dans lequel chaque processus voit les écritures respecte un *ordre causal* contenant non seulement l'ordre des processus, mais également de la *relation d'écriture-lecture* : l'écriture d'une valeur dans un registre doit précéder la lecture de cette valeur dans ce registre selon l'ordre causal. La mémoire causale est définie formellement sur la figure 2.24.

La mémoire causale peut être implémentée dans les systèmes sans-attente en utilisant la réception causale (voir page 2.1.2). Un algorithme optimal est présenté dans [17]. La réplication partielle peut être utilisée pour permettre aux processus qui n'accèdent jamais à certains registres de réduire l'espace nécessaire dans leur mémoire locale [55]. La mémoire causale est très coûteuse à implémenter dans les systèmes à très grande échelle car la taille de l'information de contrôle qui doit nécessairement être ajoutée dans les messages est linéaire par rapport au nombre de processus [32]. Des travaux [15, 120] ont cherché à réduire la complexité en laissant l'application choisir pour quels registres les liens causaux sont importants.

Définir la cohérence causale indépendamment du type de données abstrait est un problème compliqué. En effet, la relation d'écriture-lecture joue un rôle central dans la définition de la mémoire causale et elle est très liée à la notion même de registre. Plusieurs travaux ont tenté l'extension de la cohérence causale aux autres types de données pour la conjuguer à la convergence. La plupart d'entre eux [40, 77, 117] supposent simplement que l'implémentation utilise la réception causale. Cette approche n'est pas satisfaisante car elle est complètement dépendante du système. Dans [13], Attiya, Ellen et Morrison définissent la cohérence causale comme la convergence forte dans laquelle la relation de visibilité est transitive. Cependant, un tel critère appliqué à la mémoire ne donne pas la mémoire causale qui n'impose pas la convergence. Nous étudierons ce problème plus en détails dans le chapitre 4.

Autres modèles de mémoire

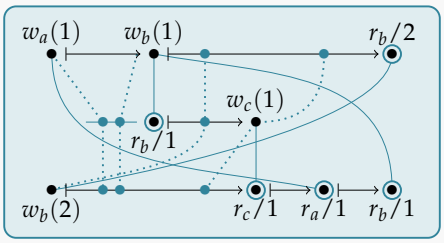
Dans la *cohérence causale temps-réel* (« Real Time Causal Consistency ») [78], l'ordre causal est défini de la même manière que dans la mémoire causale, avec la contrainte supplémentaire qu'une écriture ne peut pas précéder une lecture dans l'ordre causal et la succéder dans le temps réel. L'ordre causal ainsi obtenu est ensuite utilisé de la même manière que la relation de visibilité dans la convergence forte. Les auteurs prouvent que ce critère peut être implémenté dans les systèmes sans attente et qu'aucun critère strictement plus fort ne peut y être implémenté. Cependant, la preuve ne considère que les implémentations dans lesquels exactement un message est diffusé par écriture et aucun par lecture, ce qui limite la portée de ce résultat.

Dans les réseaux pair-à-pair, il est très coûteux de garantir que tous les processus recevront un message diffusé. Il est donc possible qu'après une écriture, tous ceux qui ont reçu la valeur quittent le système et que la valeur écrite soit perdue. La mémoire causale faiblement persistante [16, 80] est une version affaiblie de la mémoire causale qui impose seulement aux linéarisations des processus de contenir les écritures écrites une infinité de fois.

De nombreux modèles de mémoire sont construits en hybridant plusieurs des modèles précédents, chaque critère étant garanti dans des situations différentes. Dans la *cohérence faible* (« weak consistency ») [38], tous les registres ne jouent pas le même rôle. Certains registres sont des variables de synchronisation séquentiellement cohérentes. La séquence des opérations sur ces variables fixe un cadre dans lesquelles viennent

MÉMOIRE CAUSALE

La définition de la mémoire causale (définition 2.33) construit explicitement un ordre causal composé de l'ordre de processus et d'une relation d'écriture-lecture *(« writes-into order ») $\multimap$ qui relie chaque lecture à l'écriture qui a écrit la valeur lue (définition 2.32). Dans l'exemple ci-contre, la relation d'écriture-lecture est marquée par des traits pleins terminés par des cercles entourant la lecture. Si l'union de l'ordre de processus et de la relation d'écriture-lecture est acyclique, on peut construire un ordre causal $\xrightarrow{CM}$ la contenant. Soit X un ensemble de noms de registres. Une histoire H est $\mathcal{M}_X$-*causale *si, comme pour la cohérence pipeline, il est possible de construire, pour chaque processus p une linéarisation contenant les lectures de p et toutes les écritures de E_H, qui respecte l'ordre causal et la spécification séquentielle de la mémoire, c'est à dire une linéarisation de* $\text{lin}\left(H^{\xrightarrow{CM}}[E_H/p]\right) \cap L(\mathcal{M}_X)$*. Dans l'exemple, les trois linéarisations sont modélisées par les successions de points bleus et noirs pour les trois processus.*



Définition 2.32 (Relation d'écriture-lecture)**.** Soit X un ensemble dénombrable de noms de registres et $H = (\Sigma, E, \Lambda, \mapsto, \nearrow)$ une histoire concurrente. Une relation $\multimap$ est une *relation d'écriture-lecture* si les trois propriétés suivantes sont réunies.

- Une relation d'écriture-lecture lie des écritures aux lectures de la même valeur sur le même registre (par exemple $\mathrm{w}_a(1) \multimap \mathrm{r}_a/1$) :

$$\forall e, e' \in E_H, \exists x \in X, \exists n \in \mathbb{N},\ e \multimap e' \Rightarrow \Lambda(e) = \mathrm{w}_x(n) \wedge \Lambda(e') = \mathrm{r}_x/n$$

- Une lecture ne peut être reliée qu'à une seule écriture :

$$\forall e \in E_H : \left|\lfloor e \rfloor_{\multimap}\right| \leq 1$$

- Une lecture qui n'est pas reliée à une écriture retourne la valeur initiale 0 :

$$\forall e \in E_H, \exists x \in X, \exists n \in \mathbb{N},\ (\Lambda(e) = \mathrm{r}_x/n \wedge \lfloor e \rfloor_{\multimap} = \emptyset) \implies n = 0$$

Définition 2.33 (Mémoire causale)**.** Soit X un ensemble dénombrable de noms de registres. Une histoire concurrente H est $\mathcal{M}_X$*-causale* s'il existe une relation d'écriture-lecture $\multimap$ telle que :

- il existe un ordre partiel $\xrightarrow{CM}$ qui contient $\multimap$ et $\mapsto$,
- $\forall p \in \mathscr{P}_H, \text{lin}\left(H^{\xrightarrow{CM}}[E_H, p]\right) \cap L(\mathcal{M}_X) \neq \emptyset$.

L'histoire ci-contre n'est pas $\mathcal{M}_{[a-z]}$-causale. En effet, comme la première lecture du deuxième processus retourne une valeur non nulle, elle doit avoir un antécédent selon la relation d'écriture-lecture. Cet antécédent ne peut être que l'événement étiqueté $w_b(1)$, seule écriture sur b. De même, l'écriture $w_c(1)$ précède la lecture $r_c/1$. Ainsi, pour que la linéarisation demandée pour le troisième processus respecte l'ordre causal, l'écriture $w_a(1)$ doit précéder la lecture $r_a/0$ et aucune écriture $w_a(0)$ ne peut s'intercaler entre les deux, ce qui va à l'encontre de la spécification séquentielle.

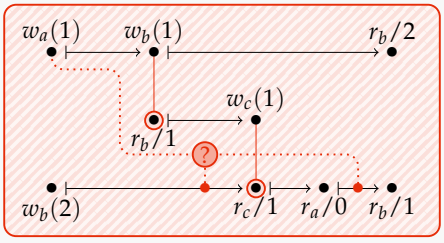


FIGURE 2.24 – La mémoire causale.

s'inscrire les opérations sur les autres variables. Soient x et y des variables de synchronisation et z un registre quelconque utilisés par deux processus p_1 et p_2. Si p_1 écrit dans z puis accède à x tandis que p_2 accède à y puis lit z, et l'accès à x précède celui à y dans la linéarisation, alors la lecture de z par p_2 retournera une valeur au moins aussi récente que celle écrite par p_1. La cohérence relâchée (« released consistency ») [46] et la cohérence d'entrée (« entry consistency ») [20] sont des variantes de la cohérence faible.

La *cohérence fish-eye* [44] se base sur une *relation de proximité* entre les nœuds d'un réseau. Par exemple, les principaux services de l'informatique en nuage exploitent plusieurs centres de données répartis autour du monde. Les ordinateurs d'un même centre de données peuvent communiquer entre eux plus efficacement que des ordinateurs de centres différents. Ils sont donc reliés par la relation de proximité. La cohérence fish-eye garantit la cohérence séquentielle entre les nœuds proches et la mémoire causale entre les nœuds éloignés. Cette idée rejoint l'algorithme proposé dans [62] qui s'adapte automatiquement aux problèmes qui peuvent se produire dans le système : en temps normal, la cohérence séquentielle est implémentée, mais en cas de partition, un service minimal est assuré pour garantir la mémoire causale.

Conclusion

Dans ce chapitre, nous avons vu comment modéliser des objets partagés. Nous proposons de séparer la spécification en deux facettes distinctes. D'un côté, la spécification séquentielle décrit comment se comporte l'objet dans un système séquentiel. Nous optons pour une définition à base de systèmes de transitions, car il s'agit d'un outil à la fois intuitif, très présent dans de nombreuses composantes de l'informatique, et dont l'utilisation par les outils de modélisation et de vérification est largement documentée. D'un autre côté, nous développons le concept de critère de cohérence qui exprime la façon dont la concurrence est prise en compte par l'objet partagé.

Parmi les notions existantes que nous avons pu exprimer dans notre modèle, quatre nous paraissent particulièrement importantes : la cohérence séquentielle, la sérialisabilité, la convergence, la cohérence pipeline. Remarquons que notre modèle permet une meilleure abstraction des systèmes pour lesquels les critères ont été initialement proposés. Par exemple, notre définition de la sérialisabilité s'applique aussi bien aux systèmes transactionnels qu'aux accès aux sites internet en considérant les déconnexions temporaires. Deux zones d'ombre restent cependant à clarifier.

La spécification des objets convergents. La convergence a une définition très différente des autres critères présentés dans ce chapitre. En fait, la spécification séquentielle $L(T)$ n'est pas du tout utilisée dans sa définition, ce qui rend ce critère extrêmement faible. C'est cette particularité qui a rendu les spécifications concurrentes nécessaires. La cohérence d'écritures décrite dans le chapitre 3 prend en compte la spécification séquentielle, ce qui renforce énormément la convergence et répond au problème de spécification de l'état de convergence.

La cohérence causale. La définition de la mémoire causale contient en elle-même la définition des liens qui unissent les lectures et les écritures. La notion de relation d'écriture-lecture n'a pas d'expression naturelle dans notre formalisme. Pour cette raison, étendre la cohérence causale à tous les types de données abstraits est un problème difficile. Nous traitons ce problème en détails dans le chapitre 4.

L'histoire de la linéarisabilité nous conforte dans notre démarche consistant à définir les critères de cohérence indépendamment des types de données abstraits. Comme beaucoup de critères faibles, elle était initialement uniquement une spécification de la mémoire partagée : le registre atomique. Son extension aux autres types de données abstraits en a fait un outil central dans l'étude des systèmes répartis, en particulier en ce qui concerne la calculabilité dans les systèmes répartis. Nous étudions l'impact de notre approche sur l'étude de la calculabilité dans les systèmes sans-attente dans le chapitre 5.

La question de la composabilité pose problème. Nous avons vu en effet que très peu de critères étaient naturellement composables, et les critères composables sur le chemin de la cohérence séquentielle sont en fait très faibles. Une autre approche est donc nécessaire pour développer des applications utilisant plusieurs objets. Dans le chapitre 6, nous décrivons une bibliothèque permettant l'application automatique de critères de cohérence à des types de données définis séquentiellement. Ainsi, l'algorithme peut composer les spécifications séquentielles avant d'appliquer le critère de cohérence plutôt que d'appliquer le critère à chacun des composés séparément.

3

La cohérence d'écritures

Sommaire

Préambule

Aucun des trois scénarios envisagés dans l'introduction ne décrit exactement ce qui se passa entre Alice et Bob ce jour là. En vérité, ils n'utilisaient ni Hangouts, ni WhatsApp, ni même Skype, mais Facebook Messenger[1] (leur conversation est retranscrite sur la figure 3.1). Tout commença comme nous l'avions imaginé dans le cas de WhatsApp : Bob ne se rendit pas compte qu'Alice avait interprété son « Évidemment » comme une réponse à « tu m'en veux ? » et non à « pause café ? », et il eut besoin de tout son pouvoir de conviction, plus tard, pour remettre les choses en ordre avec Alice. Pourtant, quand ils en reparlèrent le soir, Bob fut incapable de se justifier : l'ordre des messages avait

[1] https://www.messenger.com/

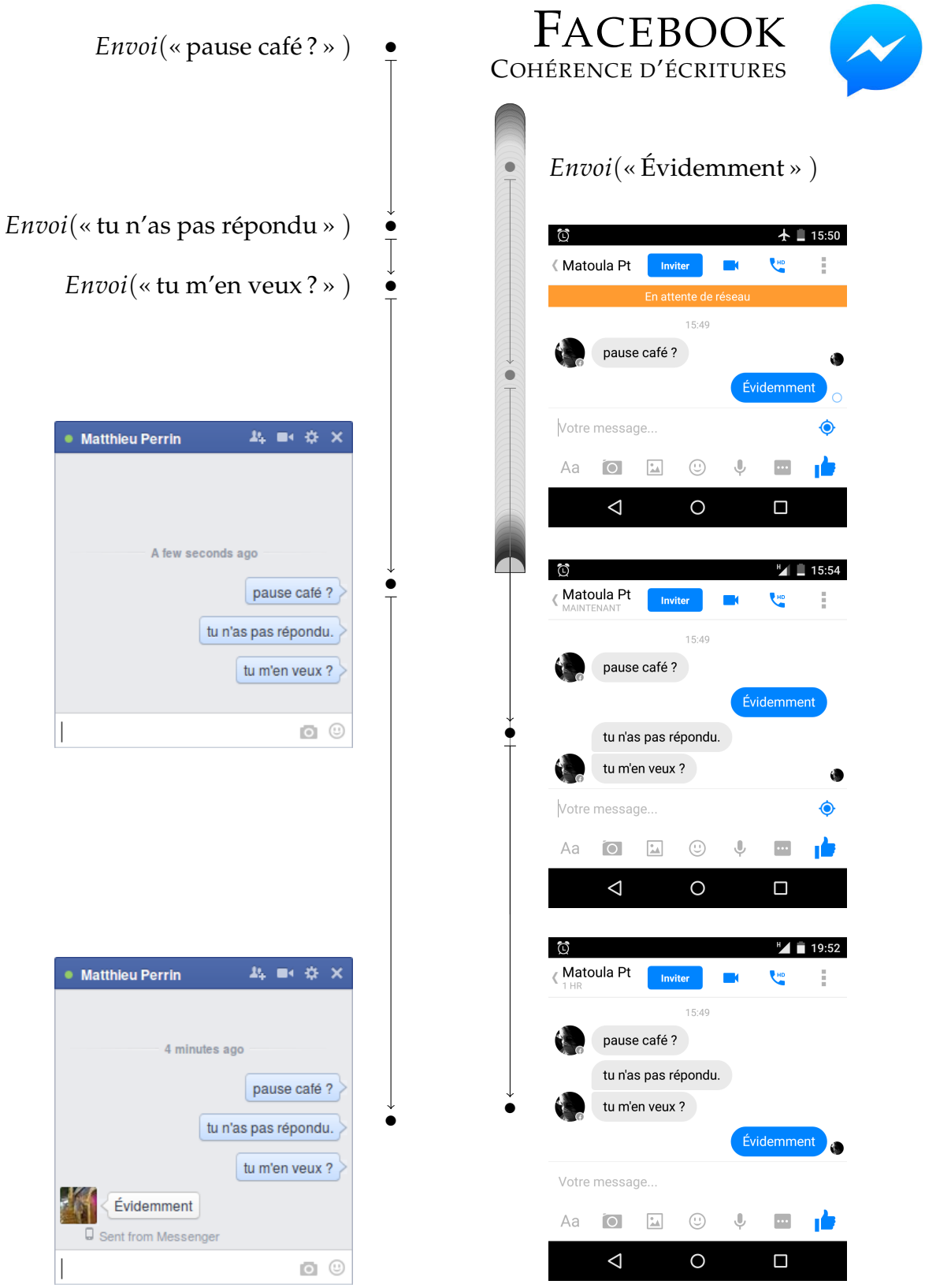


FIGURE 3.1 – Comportement de Facebook Messenger après une déconnexion.

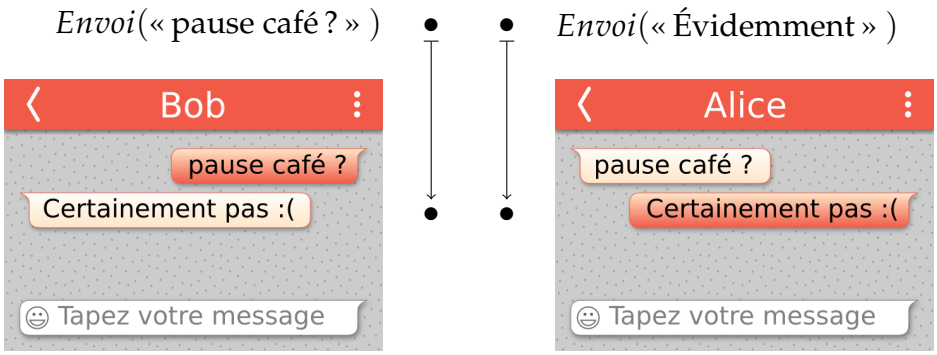


FIGURE 3.2 – Une histoire ne vérifiant pas la cohérence d'écritures.

changé depuis l'après-midi, si bien que son fil de messages était désormais le même que celui d'Alice.

La joie des retrouvailles effaça promptement les inquiétudes de chacun, d'autant que leur mésaventure leur rappelait une blague de Carole, leur amie commune, au premier avril précédent : elle leur présenta sa propre application de messagerie instantanée qui remplaçait aléatoirement les messages envoyés. Alice avait également proposé un café à Bob qui avait répondu par son « Évidemment » habituel, mais le message « Certainement pas :(» fut transmis. Heureusement, aucune conséquence ne fut à déplorer cette fois là, Bob ayant pu voir la transformation du message (les images qui apparaissent dans la retranscription de leur conversation sur la figure 3.2 ne sont heureusement pas issues d'une expérience réelle[2]).

Introduction

Les histoires observées générées par Skype (page 13), Facebook Messenger et la blague du premier avril sont toutes les trois convergentes (voir page 54) : lors de leur dernière lecture, les applications de Bob et d'Alice ont bien retourné les mêmes messages dans le même ordre. Pourtant, ces trois applications offrent des expériences utilisateur bien différentes.

La question qui se pose est naturellement la suivante.

Problème. *Comment spécifier les objets partagés convergents ?*

Dans l'histoire de la conversation par Facebook Messenger sur la figure 3.1, la convergence se fait au bout d'un temps très long. La convergence forte (voir page 56) est faite pour empêcher cette situation : lors de ses deuxième et troisième lectures, Alice a reçu les mêmes messages de Bob, elle devrait donc lire le même état. L'apport de la convergence forte par rapport à la convergence est en fait trop faible pour jouer ce rôle. En effet, le nombre de passés visibles possibles augmente exponentiellement avec le nombre d'écritures dans l'histoire. Dans les histoires convergentes, plus le nombre d'écritures augmente, plus il est facile de trouver un passé visible différent pour toutes les lectures faites avant convergence. Dans cette histoire, on peut imaginer que le passé visible de la deuxième lecture de Bob ne contient que les deux premiers envois d'Alice

[2]Le jour de la soutenance de cette thèse, Facebook a annoncé la correction d'une faille de sécurité dans Messenger permettant de produire ce type d'histoire [61].

et le sien, alors que le passé visible de sa troisième lecture contient tous les envois. Comme les deux dernières lectures de Bob n'ont pas le même passé visible, il est acceptable qu'elles retournent des résultats différents.

Le biais utilisé dans le raisonnement précédent est qu'il est contre-intuitif que Bob puisse lire le dernier message d'Alice dans sa deuxième lecture si l'envoi de ce message ne fait pas partie du passé visible de cette lecture ; rien n'empêche pourtant cela dans la définition. Ce manque force à utiliser des techniques complémentaires de spécification, comme les spécifications concurrentes. Cette approche ne fait que reporter le problème de la spécification sur les spécifications concurrentes : il est par exemple possible de proposer une spécification concurrente du service de messagerie de la figure 3.2 qui indique que si un processus a connaissance d'un message « pause café ? » suivi d'un message « Évidemment », il doit être dans l'état [« pause café ? »; « Certainement pas :(»]. Dès lors, pourquoi Alice et Bob se plaindraient-ils de ce service de messagerie instantanée ? Nous avons déjà discuté plus en détails des limites des spécifications concurrentes dans la partie 2.2.3.

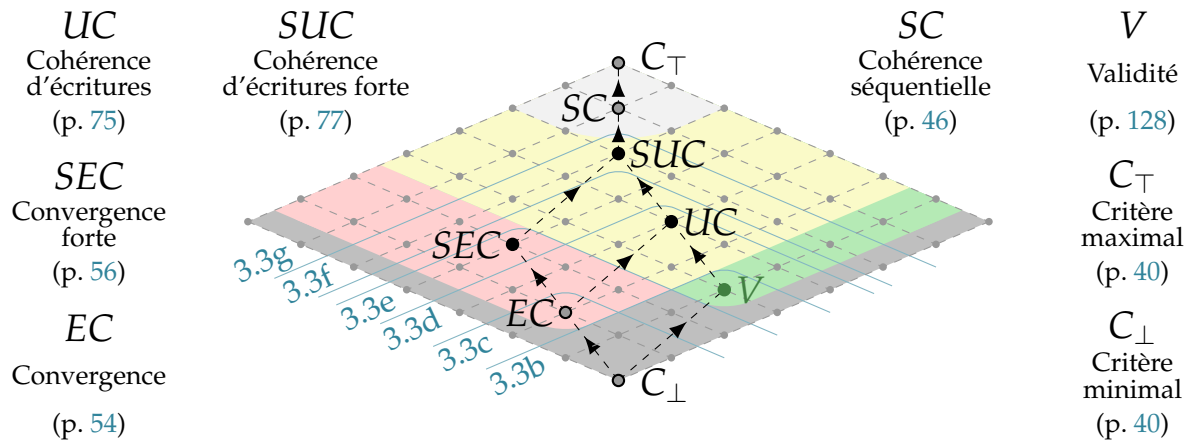
Approche. *Nous proposons de rendre leur place aux spécifications séquentielles en ajoutant une contrainte à la convergence : l'état de convergence doit être compatible avec une exécution séquentielle de toutes les écritures de l'histoire. Nous appelons* cohérence d'écritures *ce nouveau critère de cohérence et introduisons la* cohérence d'écritures forte *comme l'équivalent pour la cohérence d'écritures, de la convergence forte pour la convergence.*

Une question importante pour justifier l'introduction de ces nouveaux critères est celle de leur implémentation. Nous donnons une place particulière à cette question dans ce chapitre en étudiant trois algorithmes génériques offrant des complexités différentes. Le premier est efficace en considération du nombre de messages échangés, mais requiert une quantité de mémoire et de puissance de calcul non bornées. Le deuxième résout ce problème en augmentant le nombre de messages échangés. Le troisième est paramétré pour achever un compromis permettant d'exploiter au mieux la mémoire et la capacité du réseau. La définition des nouveaux critères de cohérence a été présentée sous forme d'un article court à DISC 2014 [92], puis le contenu de la section 3.1 et l'existence de l'implémentation générique de la cohérence d'écritures forte ont été présentés à IPDPS 2015 [93]. Le dernier algorithme présenté a été mis au point en collaboration avec Olivier Ruas [104].

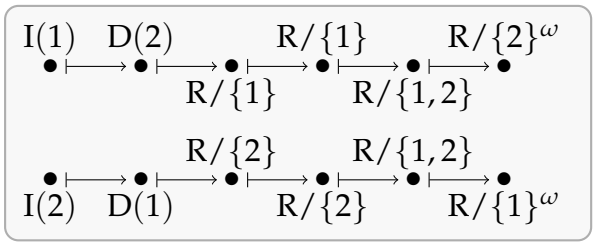
Contributions. *Pour résumer, nous présentons deux contributions majeures dans ce chapitre.*

1. *Deux nouveaux critères de cohérence, la cohérence d'écritures et la cohérence d'écritures forte, que nous comparons avec la convergence et la convergence forte.*
2. *Trois nouveaux algorithmes génériques pour les implémenter.*

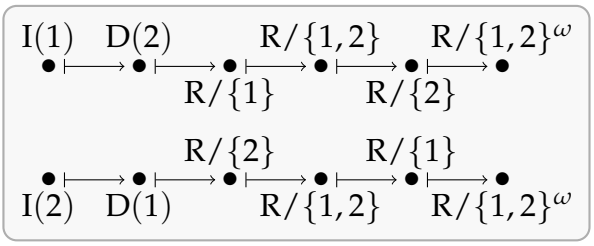
Ce chapitre est organisé comme ceci. La partie 3.1 présente les deux nouveaux critères : la cohérence d'écritures et la cohérence d'écritures forte, et les compare à la convergence et la convergence forte. La partie 3.2 présente trois algorithmes pouvant être adapté pour implémenter tous les types de données abstraits. La partie 3.3 présente la démonstration de la correction du troisième d'entre eux et des expériences visant à mesurer sa complexité.



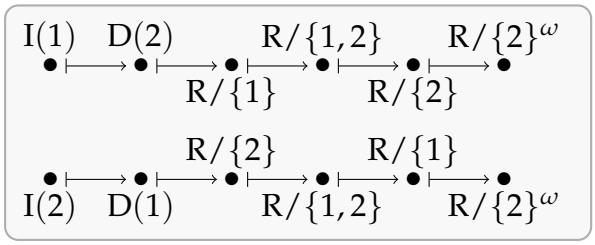
(a) Force relative de la cohérence d'écritures.



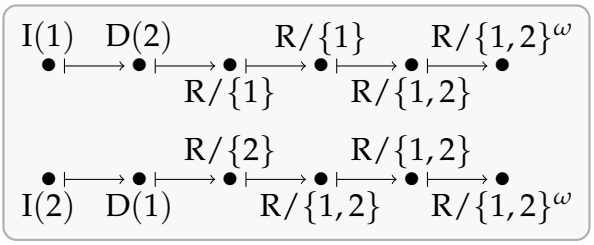
(b) Pas *EC*



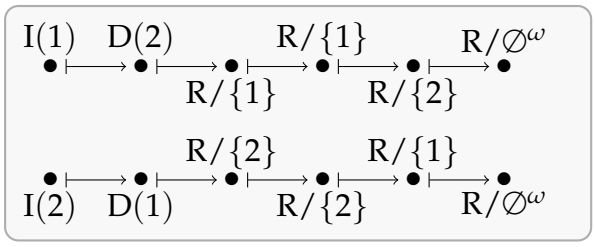
(c) *EC* mais pas *UC* ni *SEC*



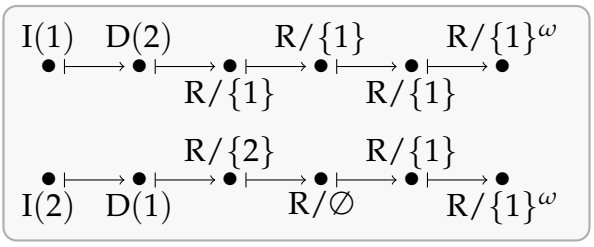
(d) *UC* mais pas *SEC*



(e) *SEC* mais pas *UC*

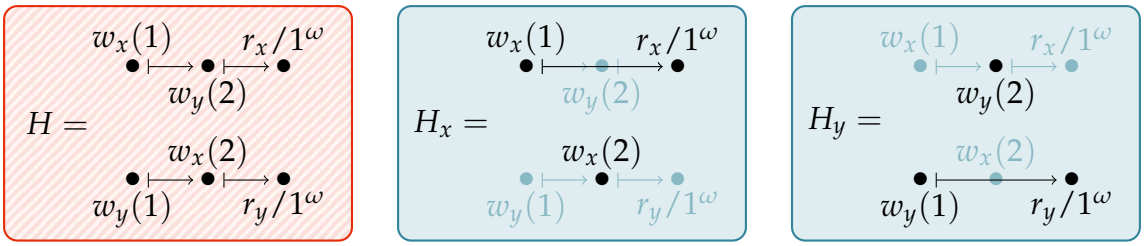


(f) *UC* et *SEC* mais pas *SUC*

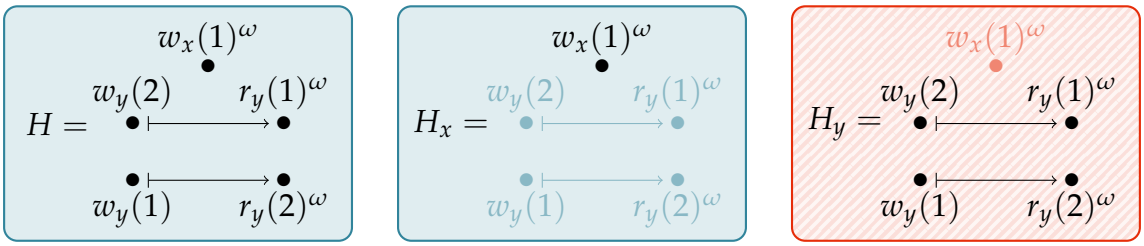


(g) *SUC* mais pas *SC*

FIGURE 3.3 – Deux processus partagent une instance de l'ensemble d'entiers ($\mathcal{S}_{\mathbb{N}}$). Dans toutes les histoires, le premier processus insère la valeur 1 (opération I(1)), supprime la valeur 2 (opération D(2)) puis lit une infinité de fois (opération R). Le second processus insère la valeur 2, supprime la valeur 1 et lit une infinité de fois.



(a) Absence de composabilité : $H_x \in UC(\mathcal{M}_x)$ et $H_y \in UC(\mathcal{M}_y)$ mais $H \notin UC(\mathcal{M}_{\{x,y\}})$



(b) Absence de décomposabilité : $H \in UC(\mathcal{M}_{\{x,y\}})$ mais $H_y \notin UC(\mathcal{M}_y)$

FIGURE 3.4 – La cohérence d'écritures n'est ni composable ni décomposable.

3.1 La cohérence d'écritures

3.1.1 Cohérence d'écritures et cohérence d'écritures forte

Dans cette partie, nous introduisons deux nouveaux critères : la cohérence d'écritures et la cohérence d'écritures forte et nous les comparons à la convergence et la convergence forte (définies formellement dans le chapitre 2).

La cohérence d'écritures

La cohérence d'écritures (*UC*, pour « Update Consistency »), formellement définie sur la figure 3.5, est le critère de cohérence qui permet de discriminer l'histoire de la blague de Carole sur la figure 3.2. Le problème dans cette histoire est que l'état de convergence ne reflète pas les écritures. La cohérence d'écritures renforce la convergence en prenant en compte la spécification séquentielle du type de données abstrait. Plus précisément, elle impose, outre la convergence, que l'état de convergence soit l'un des états autorisés par la cohérence séquentielle. Ainsi, il existe un ordre total sur toutes les écritures (d'où le nom du critère) qui respecte l'ordre de programme tel que l'exécution de toutes les écritures dans cet ordre mène à l'état de convergence.

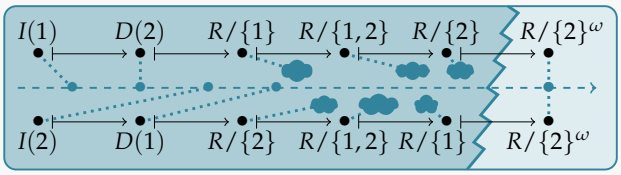
En supposant que la dernière écriture de chaque processus est répétée une infinité de fois, l'histoire de la figure 3.2 ne vérifie pas la cohérence d'écritures. En effet, comme il n'existe que deux façons d'ordonner deux écritures, les seuls états de convergence possibles sont [« pause café ? »; « Évidemment »] et [« Évidemment »; « pause café ? »]. Aucun d'entre eux ne correspondant à l'état lu, l'histoire ne vérifie pas la cohérence d'écritures.

La cohérence d'écritures n'est ni composable ni décomposable, comme le montre la figure 3.4. Sur la figure 3.4a, deux processus partagent deux registres nommés x et y. Le scénario est très semblable aux histoires de la figure 3.3 : chaque processus écrit 1 dans

COHÉRENCE D'ÉCRITURES

✗ Non composable p. 41
✗ Non décomposable p. 41
✓ Faible p. 126

Comme la convergence, la cohérence d'écritures impose que si tous les participants arrêtent d'écrire, tous les réplicas finissent par converger vers un état commun. De plus, les états de convergence possibles sont exactement ceux que l'on peut atteindre avec la cohérence séquentielle : il doivent résulter d'une linéarisation des écritures. Par exemple, dans l'histoire de la figure 3.3d (rappelée ci-contre), au bout d'un moment, toutes les lectures retournent la valeur $\{2\}$ qui peut être obtenue avec la séquence $I(1) \cdot D(2) \cdot I(2) \cdot D(1)$.



Formellement, une histoire H respecte la cohérence d'écritures pour un type de données abstrait T (définition 3.1) si elle rentre dans l'un des deux cas suivants.

- *H contient une infinité d'écritures (c'est-à-dire $|U_{T,H}| = \infty$). Dans ce cas, au moins un participant ne s'arrête jamais d'écrire, donc les valeurs retournées par les lectures peuvent continuer à évoluer en fonction des écritures.*
- *Il est possible de supprimer un nombre fini de lectures (celles faites avant convergence) pour obtenir une histoire séquentiellement cohérente. Dans la définition 3.1, ces lectures sont contenues dans l'ensemble $E_H \setminus E'$ (qui doit être fini) et l'histoire $H[E'/E_H]$ contient toutes les écritures (dans E_H) et les lectures faites après convergence (qui appartiennent à E').*

Dans l'exemple ci-dessus, l'une des valeurs possibles pour E' contient uniquement les lectures retournant $\{2\}$ à la fin, et la séquence suivante est une linéarisation possible de $H[E'/E_H]$. Notons que les symboles d'entrée des lectures R sont bien présentes dans la linéarisation car les opérations sont seulement cachées, mais ils ne jouent aucun rôle dans la spécification séquentielle puisqu'ils correspondent à des boucles sur les états.

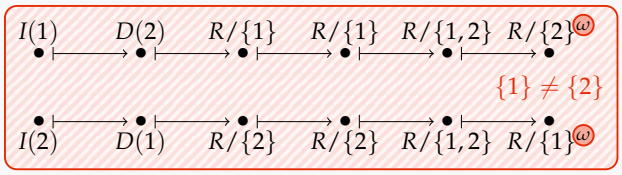
$$I(1) \cdot D(2) \cdot I(2) \cdot D(1) \cdot R \cdot R \cdot R \cdot R \cdot R \cdot R \cdot R/\{2\}^\omega$$

Définition 3.1 (Cohérence d'écritures)**.** La *cohérence d'écritures* est le critère de cohérence

$$UC : \left\{ \begin{array}{lll} \mathcal{T} & \to & \mathcal{P}(\mathcal{H}) \\ T & \mapsto & \left\{ H \in \mathcal{H} : \begin{array}{rl} & |U_{T,H}| = \infty \\ \vee & \exists E' \subset E_H, \quad |E_H \setminus E'| < \infty \\ & \qquad \wedge \quad \mathrm{lin}(H[E'/E_H]) \cap L(T) \neq \emptyset \end{array} \right\} \end{array} \right.$$

La cohérence d'écritures garantit deux propriétés.

1. La convergence est atteinte. *L'histoire de la figure 3.3b ne satisfait pas la cohérence d'écritures car une infinité de lectures retournent 1 et une autre infinité de lectures retournent 2, bien que l'histoire ne contienne que six écritures.*



2. L'ordre de processus entre les écritures est respecté. *L'histoire de la figure 3.3c ne satisfait pas non plus la cohérence d'écritures. En effet, toutes les lectures faites après convergence le sont dans l'état $\{1,2\}$. Or, dans toute linéarisation, la dernière écriture est une suppression. Les seuls états de convergence admissibles pour cette histoire sont $\emptyset$, $\{1\}$ et $\{2\}$.*

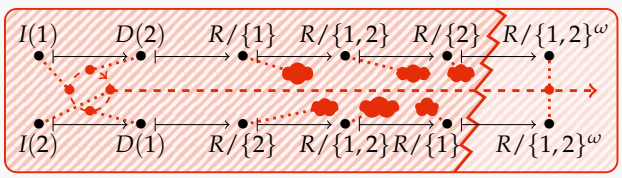


FIGURE 3.5 – La cohérence d'écritures.

l'un des registres puis 2 dans l'autre. La convergence vers l'état dans lequel les deux registres valent 1 est interdite par la cohérence d'écritures. Pourtant en considérant chaque registre séparément, l'écriture du 2 pourrait être placée avant celle du 1. Ainsi, les deux sous-histoires H_x et H_y vérifient la cohérence d'écritures, mais pas l'histoire H. La figure 3.4b montre que la cohérence d'écritures n'est pas décomposable en utilisant la même technique que pour la convergence (page 55) : l'histoire proposée comporte une infinité d'écritures sur x, ce qui est suffisant pour garantir la cohérence d'écritures indépendamment des lectures faites sur y.

Proposition 3.1 ($EC \leq UC$)**.** *La cohérence d'écritures renforce la convergence.*

Démonstration. Soient un type de données abstrait $T = (\mathsf{A}, \mathsf{B}, \mathsf{Z}, \zeta_0, \tau, \delta)$ et une histoire concurrente $H = (\Sigma, E, \Lambda, \mapsto, \nearrow)$ qui vérifie la cohérence d'écritures par rapport à T (c'est-à-dire $H \in UC(T)$).Montrons que H est convergente (c'est-à-dire $H \in EC(T)$).

Si H contient une infinité d'écritures ou un nombre fini de lectures pures, elle est convergente par définition. Supposons que H contient un nombre fini d'écritures et une infinité de lectures pures. Comme H vérifie la cohérence d'écritures, il existe un ensemble cofini d'événements $E' \subset E_H$ et un mot $l \in \text{lin}(H[E_H/E']) \cap L(T)$. Comme le nombre d'écritures est fini, il existe un préfixe fini l' de l qui les contient toutes. Comme $l' \in L(T)$, il étiquette un chemin entre ζ_0 et un certain état ζ dans l'ADT. Tous les événements qui sont dans l mais pas dans l' sont des lectures pures, donc ils sont tous effectués dans l'état ζ, d'où $H \in EC(T)$. □

La cohérence d'écritures forte

Si l'histoire de la blague de Carole (figure 3.2) contient un nombre fini de lectures, en revanche, elle vérifie à la fois la convergence forte et la cohérence d'écritures, mais pour des raisons différentes : la convergence forte parce que la spécification séquentielle n'est pas prise en compte et la cohérence d'écritures parce que l'instant de convergence n'est pas encore atteint. La cohérence d'écritures forte (*SUC*, pour « Strong Update Consistency »), définie formellement sur la figure 3.6, est à la cohérence d'écritures ce que la convergence forte est à la convergence. Comme la convergence forte, la cohérence d'écritures forte impose que deux opérations ayant le même passé selon la relation de visibilité soient faites dans le même état, et comme dans la cohérence d'écritures, cet état doit être le résultat d'une linéarisation des écritures de leur passé commun. Dans l'histoire de la figure 3.2, quel que soit son passé visible parmi les deux envois de messages, jamais une lecture ne pourra retourner le message « Certainement pas :(».

Proposition 3.2 ($UC \leq SUC$)**.** *La cohérence d'écritures forte renforce la cohérence d'écritures.*

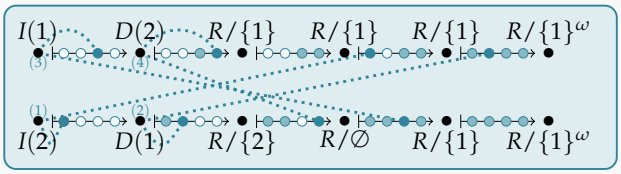
Démonstration. Soient un type de données abstrait $T = (\mathsf{A}, \mathsf{B}, \mathsf{Z}, \zeta_0, \tau, \delta)$ et une histoire concurrente $H = (\Sigma, E, \Lambda, \mapsto, \nearrow)$ qui vérifie la cohérence d'écritures forte par rapport à T (c'est-à-dire $H \in SUC(T)$). Montrons que H vérifie la cohérence d'écritures (c'est-à-dire $H \in UC(T)$).

Si H contient une infinité d'écritures ou un nombre fini de lectures pures, elle vérifie la cohérence d'écritures par définition. Supposons que H contient un nombre fini d'écritures et une infinité de lectures pures. Comme H vérifie la cohérence d'écritures forte, il existe une relation de visibilité $\xrightarrow{\text{VIS}}$ et un ordre total $\leq$ qui contient $\xrightarrow{\text{VIS}}$ tels que pour tout $e \in E_H$, il existe une linéarisation $l_e \cdot \Lambda(e) \in \text{lin}\left(H^{\leq}[\lfloor e \rfloor_{\xrightarrow{\text{VIS}}} / \{e\}]\right) \cap L(T)$.

COHÉRENCE D'ÉCRITURES FORTE

✗ Non composable p. 41
✗ Non décomposable p. 41
✓ Faible p. 126

La cohérence d'écritures forte est à la cohérence d'écritures ce que la convergence forte est à la convergence. Comme la convergence forte, la cohérence d'écritures forte impose que deux opérations ayant la même visibilité soient faites dans le même état. Comme dans la cohérence d'écritures, les opérations d'écritures sont totalement ordonnées. L'état lu pour tout événement résulte de l'exécution des écritures de son passé selon la relation de visibilité, ordonnées selon l'ordre total. Dans l'histoire de la figure 3.3g rappelée ci-contre, la relation de visibilité est figurée par des pointillés et l'ordre total par l'ordre des quatre points qui précèdent chaque opération. Les lectures faites avant la convergence sont le résultat d'une sous-séquence de la séquence des écritures, les « trous » (figurés par des ronds blancs) étant comblés au fur et à mesure de l'exécution.



Formellement, une histoire H vérifie la cohérence d'écritures forte pour un type de données abstrait T (définition 3.1) s'il existe une relation de visibilité $\xrightarrow{\text{VIS}}$ (qui modélise la réception des messages, voir page 56) et un ordre total $\leq$ sur les événements de H qui contient $\xrightarrow{\text{VIS}}$ tels que l'histoire $\left(H^{\leq}[\lfloor e \rfloor_{\xrightarrow{\text{VIS}}} / \{e\}]\right)$, qui contient les écritures du passé de e selon $\xrightarrow{\text{VIS}}$ ordonnées selon $\leq$ et e comme unique lecture potentielle, est séquentiellement cohérente. Dans l'exemple ci-dessus, l'ordre total est tel que $I(2) \leq D(1) \leq I(1) \leq D(2)$ (en confondant les événements et leurs étiquettes) et les séquences suivantes sont les linéarisations pour le deuxième processus (en ignorant les lectures) :

$$I(2) \cdot D(1) \cdot R/\{2\} \qquad I(2) \cdot D(1) \cdot D(2) \cdot R/\emptyset \qquad I(2) \cdot D(1) \cdot I(1) \cdot D(2) \cdot R/\{1\}$$

Définition 3.2 (Cohérence d'écritures forte)**.** La *cohérence d'écritures forte* est le critère de cohérence

$$SUC : \left\{ \begin{array}{lll} T & \to & \mathcal{P}(\mathcal{H}) \\ T & \mapsto & \left\{ H \in \mathcal{H} : \begin{array}{l} \exists \xrightarrow{\text{VIS}} \in Vis(H), \exists \leq \text{ ordre total sur } E_H, \xrightarrow{\text{VIS}} \subset \leq \\ \wedge \quad \forall e \in E_H, \text{lin}\left(H^{\leq}[\lfloor e \rfloor_{\xrightarrow{\text{VIS}}} / \{e\}]\right) \cap L(T) \neq \emptyset \end{array} \right\} \end{array} \right.$$

La cohérence d'écritures forte est strictement plus forte que UC + SEC. En effet, l'histoire de la figure 3.3f (rappelée ci-contre) vérifie la cohérence d'écritures et la convergence forte mais pas la cohérence d'écritures forte. Le problème a lieu lors de la lecture du 1 par le deuxième processus. Pour que l'état de convergence soit ∅, il faut que $I(I) \leq D(1)$. Or la suppression du 1 précède la lecture du 1 selon la relation de visibilité et il est impossible qu'il y ait une autre insertion de 1 entre la suppression et la lecture.

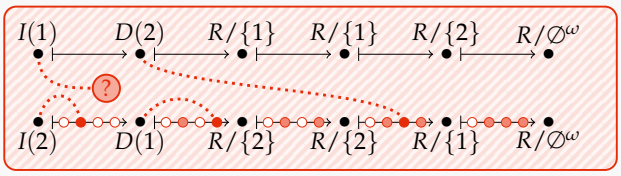


FIGURE 3.6 – La cohérence d'écritures forte.

Soit E' l'ensemble des événements qui contiennent toutes les écritures dans leur passé selon l'ordre total : $E' = \bigcap_{u \in U_{T,H}} \{e \in E_H, u < e\}$. Le complémentaire de E' est $E_H \setminus E' = \bigcup_{u \in U_{T,H}} \{e \in E_H, e \leq u\}$, c'est-à-dire une union finie (il y a un nombre fini d'écritures dans l'histoire) d'ensembles finis (d'après la troisième condition dans la définition 2.24 et le fait que $\leq$ contient $\xrightarrow{\text{VIS}}$), et est donc fini.

L'histoire $H^{\leq}[E_H/E']$ contient une unique linéarisation l, puisque l'ordre $\leq$ est total. Pour tout événement $e \in E'$, les mêmes écritures dans le même ordre sont présentes avant e dans les linéarisations l et l_e. On en déduit que $l \in L(T)$, et donc que $H \in UC(T)$. □

Proposition 3.3 ($SEC \leq SUC$)**.** *La cohérence d'écritures forte renforce la convergence forte.*

Démonstration. Soient un type de données abstrait $T = (\mathsf{A}, \mathsf{B}, \mathsf{Z}, \zeta_0, \tau, \delta)$ et une histoire concurrente $H = (\Sigma, E, \Lambda, \mapsto, \nearrow)$ qui vérifie la cohérence d'écritures forte par rapport à T (c'est-à-dire $H \in SUC(T)$). Montrons que H est fortement convergente (c'est-à-dire $H \in SEC(T)$).

Comme H vérifie la cohérence d'écritures forte, il existe une relation de visibilité $\xrightarrow{\text{VIS}}$ et un ordre total $\leq$ qui contient $\xrightarrow{\text{VIS}}$ tels que pour tout $e \in E_H$, il existe une linéarisation $l_e \cdot \Lambda(e) \in \text{lin}\left(H^{\leq}[\lfloor e \rfloor_{\xrightarrow{\text{VIS}}} / \{e\}]\right) \cap L(T)$.

Pour tout $e \in E_H$, notons $V_e = \left(U_{T,H} \cap \lfloor e \rfloor_{\xrightarrow{\text{VIS}}} \setminus \{e\}\right)$. On définit la fonction f qui a tout ensemble d'écritures de E_H $V \subset \mathcal{P}(U_{T,H})$, associe l'état obtenu en exécutant toutes les écritures de V dans l'ordre $\leq$. Cette fonction est bien définie car le système de transitions de T est déterministe.

Soit $e \in E_H$. La linéarisation l_e exécute les écritures de V_e dans l'ordre $\leq$. Elle mène donc dans l'état $f(V_e)$. On en déduit que $f(V_e) \in \delta_T^{-1}(\Lambda(e))$, donc $H \in SEC(T)$. □

3.1.2 Étude de cas : l'ensemble partagé

L'ensemble est sans doute la structure de données la plus étudiée pour la convergence. Différentes extensions des CRDT ont été proposées pour implémenter un ensemble convergent bien que les opérations d'insertion et de suppression du même élément ne commutent pas. Parmi elles, l'OR-Set est une implémentation de l'ensemble partagé spécifiée par la spécification concurrente IW-Set (voir page 60). Le but de cette section est de comparer l'OR-Set et l'ensemble partagé $\mathcal{S}_{Val}$ (voir page 27) muni de la cohérence d'écritures forte.

Dans l'OR-Set, quand une insertion et une suppression du même élément sont concurrentes, la suppression est annulée, ce qui correspond à la stratégie « l'insertion gagne ». Deux propriétés restreignent l'ensemble des histoires admises par l'OR-Set : son critère de cohérence *SEC* (toutes les histoires qu'il admet sont dans $SEC(\mathcal{S}_{Val})$) et sa spécification concurrente IW-Set (toutes les histoires de $SEC(\mathcal{S}_{Val})$ ne sont pas admises). Notre preuve que la convergence forte est plus faible que la cohérence d'écritures forte ne s'applique pas à l'OR-Set car la spécification de ce dernier est renforcée par la spécification concurrente. Nous prouvons maintenant que cela reste vrai pour l'OR-Set.

Proposition 3.4 (Comparaison entre l'OR-Set et $SUC(\mathcal{S}_{Val})$)**.** *Pour tout $H \in SUC(\mathcal{S}_{Val})$, H est fortement convergente pour l'IW-Set.*

Démonstration. Soit $H \in SUC(\mathcal{S}_{Val})$. Nous définissons la nouvelle relation $\xrightarrow{IW}$ comme suit. Pour tous $e, e' \in E_H$, $e \xrightarrow{IW} e'$ si l'une des conditions suivantes est vérifiée :

- $e \xrightarrow{\text{VIS}} e'$;
- e et e' sont deux écritures sur le même élément, c'est-à-dire $e \leq e'$ et il existe $x \in Val$ tel que $\{\Lambda(e), \Lambda(e')\} \subset \{I(x)/\bot, D(x)/\bot\}$;
- e' est une lecture et il existe une écriture e'' telle que $e \xrightarrow{IW} e''$ et $e'' \xrightarrow{IW} e'$.

La relation $\xrightarrow{IW}$ est acyclique parce qu'elle est contenue dans $\leq$ et elle est bien une relation de visibilité car $\xrightarrow{\text{VIS}} \subset \xrightarrow{IW}$. De plus, il n'est pas possible que deux écritures concernant le même élément soient concurrentes selon $\xrightarrow{IW}$ et la dernière écriture pour chaque élément est la même que selon $\leq$. Par conséquent, H possède la propriété de convergence forte pour l'IW-Set. □

3.2 Implémentations génériques

Le but de cette section est d'étudier des constructions génériques implémentant la cohérence d'écritures pour tout ADT dans les systèmes sans-attente. Outre la preuve que ce critère est bien un critère faible, cette construction apporte des techniques générales qui peuvent être combinées pour implémenter la cohérence d'écritures.

Deux stratégies sont possibles. Nous les présentons toutes les deux dans un premier temps (UC_∞ dans la partie 3.2.1 et UC_0 dans la partie 3.2.2), avant de nous concentrer sur l'algorithme paramétré $UC[k]$ qui combine les deux méthodes pour en prendre les meilleurs aspects (partie 3.2.3). Le paramètre entier k permet de privilégier l'une ou l'autre des stratégies : $UC[0]$ est une optimisation de UC_0 et le comportement de $UC[k]$ se rapproche de celui de UC_∞ quand k tend vers l'infini.

3.2.1 L'algorithme UC_∞ pour la cohérence d'écritures forte

L'algorithme de la figure 3.7 présente une implémentation de la première stratégie. Un exemple d'exécution est montré sur la figure 3.8. Les écritures sont nommées par les lettres a, b, c et d et les états sont nommés d'après la suite d'opérations qui y mène (par exemple, a fait passer de l'état ζ_0 à l'état ζ_a, b fait passer de l'état ζ_a à l'état ζ_{ab}, etc.).

L'algorithme peut être rapproché de celui de Karsenty et Beaudouin-Lafon [63] pour les objets possédant une opération *undo* annulant la dernière opération. Le principe est de construire un ordre total sur les événements acceptés par tous les participants, et de réécrire l'histoire *a posteriori* pour que tous les processus atteignent l'état correspondant à la même histoire séquentielle dès qu'ils ont reçu toutes les écritures de l'histoire concurrente. N'importe quelle stratégie pour fabriquer l'ordre total pourrait fonctionner. Dans l'algorithme 3.7, cet ordre est fabriqué à partir d'une horloge logique linéaire de Lamport [71] qui contient la relation « s'est produit avant ». L'ordre des processus est donc forcément respecté. Une horloge de Lamport est un pré-ordre total car certains événements peuvent être associés à la même horloge. Pour avoir un ordre total, les estampilles sont des paires composées d'une horloge linéaire et de l'identifiant unique du processus qui a produit l'événement.

Chaque processus p_i gère sa propre vue $\mathbf{vtime}_i$ du temps logique et la liste $\mathbf{history}_i$ des événements d'écriture estampillés connus par p_i. La liste $\mathbf{history}_i$ contient des triplets (t_j, j, α) où α représente l'événement d'écriture et (t_j, j) est l'estampille. La liste est triée selon l'ordre lexicographique sur les estampilles : $(t_j, j) < (t_k, k)$ si $(t_j < t_k)$ ou $(t_j = t_k$ et $j < k)$.

L'algorithme exécute le même code pour toutes les opérations. Cependant, la méthode `apply` possède une partie de lecture, qui peut être ignorée par les écritures pures et une partie d'écriture qui n'est pas nécessaire pour les lectures pures. Quand une écriture est effectuée localement, p_i informe les autres processus en diffusant un message de manière fiable à tous les processus (y compris lui-même). Ainsi, tous les processus finiront par connaître toutes les écritures. Quand p_i reçoit un message

```
1  algorithm UC∞(A, B, Z, ζ0, τ, δ)
2      variable vtime_i ∈ ℕ ← 0 ;                    // horloge logique linéaire
3      variable history_i ⊂ (ℕ × ℕ × A) ← ∅ ;        // liste des événements
4      operation apply (α ∈ A) ∈ B
5          variable state ∈ Z ← ζ0 ;                                  // lecture
6          for (t_j, j, α') ∈ history_i triés selon (t_j, j) do
7              state ← τ(state, α') ;                  // l'histoire est rejouée
8          end
9          if α ∈ U_T then
10             broadcast mUpdate (vtime_i + 1, i, α) ;                // écriture
11         end
12         return δ(state, α) ;
13     end
14     on receive mUpdate (t_j ∈ ℕ, j ∈ ℕ, α ∈ A)
15         vtime_i ← max(vtime_i, t_j) ;
16         history_i ← history_i ∪ {(t_j, j, α)} ;
17     end
18 end
```

FIGURE 3.7 – Algorithme générique $UC_\infty(T)$: code pour p_i

mUpdate(t_j, j, α), il met son horloge à jour et insère le triplet à sa liste **history**$_i$. Pour lire l'état courant, p_i rejoue localement toutes les écritures contenues dans sa liste dans l'ordre total convenu à l'avance, en partant de l'état initial.

Un processus n'attend jamais de messages d'un autre processus, ce qui garantit que toutes les opérations terminent même en cas de pannes des autres processus. L'algorithme de la figure 3.7 génère bien des histoires concurrentes dans toutes les exécutions. Nous montrons maintenant que ces histoires vérifient la cohérence d'écritures forte.

Proposition 3.5 (Cohérence d'écritures forte)**.** *Toutes les histoires admises par l'algorithme 3.7 pour un ADT T sont dans $SUC(T)$.*

Démonstration. Soient $T \in \mathcal{T}$ et H une histoire admise par l'algorithme 3.7. Soient deux événements $e, e' \in E_H$ invoqués par les processus p_i et $p_{i'}$ dans les états respectifs (**vtime**$_i$, **history**$_i$) et (**vtime**$_{i'}$, **history**$_{i'}$). Nous posons :

- $e \xrightarrow{\text{VIS}} e'$ si e est une écriture et $p_{i'}$ a reçu le message envoyé pendant l'exécution de e avant qu'il ne commence à exécuter e', ou bien si e est une lecture et $e \mapsto e'$. Le fait qu'un message est reçu instantanément par son émetteur permet d'affirmer que $\xrightarrow{\text{VIS}}$ est bien une relation de visibilité, au sens de la définition 2.24.
- $e \leq e'$ si **vtime**$_i <$ **vtime**$_{i'}$ ou **vtime**$_i =$ **vtime**$_{i'}$ et $i \leq i'$. Cet ordre lexicographique est total car deux opérations du même processus ont des horloges différentes et l'identifiant d'un processus est unique. De plus, si $e \xrightarrow{\text{VIS}} e'$, d'après les lignes 10 et 15, **vtime**$_{i'} \leq$ **vtime**$_i + 1$ donc $e \leq e'$. Le passé de e est fini car il contient au plus $c \times n + i$ événements.

Soit $e \in E_H$. Les lignes 6 et 7 construisent explicitement une exécution qui est dans $\text{lin}(H^{\leq}[\lfloor e \rfloor_{\xrightarrow{\text{VIS}}} / \{e\}])$ par définition de $\leq$ et $\xrightarrow{\text{VIS}}$ et dans $L(T)$ par définition de T. On a donc bien $H \in SUC(T)$. □

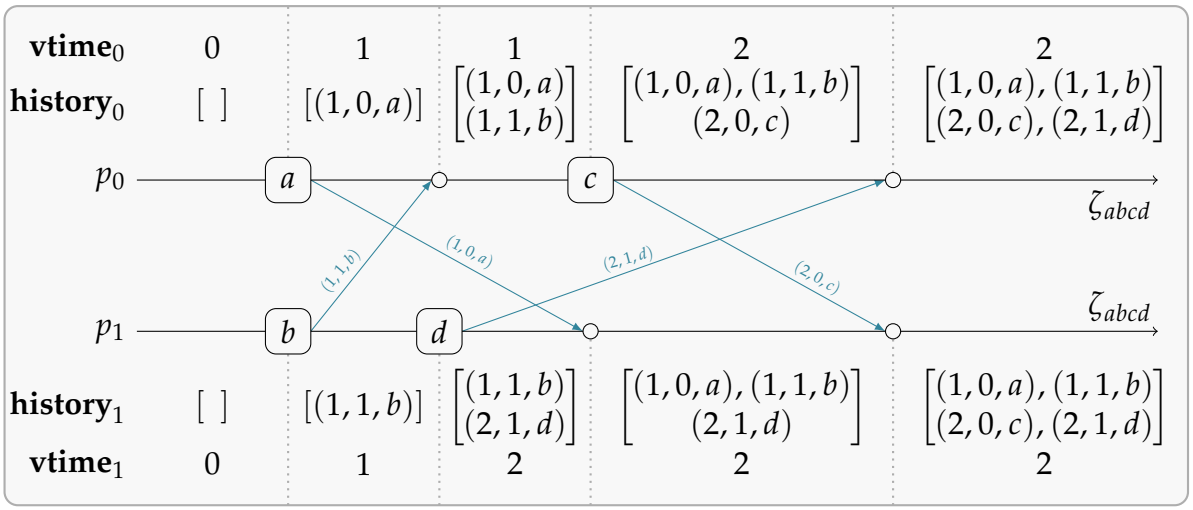


FIGURE 3.8 – Exemple d'exécution de l'algorithme UC_∞

3.2.2 L'algorithme UC_0

L'algorithme 3.9 présente la deuxième solution. Un exemple d'exécution est montré sur la figure 3.10. L'idée est que dans la cohérence pipeline, quand un processus a reçu toutes les écritures, son état est acceptable comme état de convergence selon la cohérence d'écritures. Il suffirait donc de choisir un leader dont l'état est accepté par tous les autres processus. Comme il est impossible de savoir quand toutes les opérations d'écriture ont été effectuées et quels processus sont corrects, l'identité du leader peut varier au cours du temps. Dans l'algorithme 3.9, le dernier leader est le processus correct de plus petit identifiant.

Chaque processus p_i gère trois variables. La variable $\mathbf{clock}_i$ contient une horloge vectorielle dont l'entrée $\mathbf{clock}_i[j]$ représente le nombre de messages `mUpdate` envoyés par p_j et reçus par p_i. L'état local sur lequel sont effectuées les lectures est $\mathbf{state}_i$. Enfin, si $\mathbf{leader}_i \neq i$, $\mathbf{state}_i$ contient une valeur qui a été présente dans $\mathbf{state}_{\mathbf{leader}_i}$.

Lors de l'exécution d'une opération, l'algorithme 3.9 lit son état local et diffuse un message $\texttt{mUpdate}(cl_j, j, \alpha)$ si l'opération est une écriture. Cette diffusion est FIFO pour assurer qu'un processus qui ne reçoit jamais de correction exécute toutes les opérations dans un ordre cohérent avec l'ordre de processus. Il est important ici que le message soit reçu immédiatement par p_i. Quand p_i reçoit le message de p_j, il vérifie que l'opération n'a pas déjà été prise en compte dans son état courant (auquel cas $\mathbf{clock}_i[j] \geq cl_j$), et si ce n'est pas le cas il applique l'écriture sur son état local, puis se déclare son propre leader et diffuse un message `mcorrect` contenant son horloge vectorielle, son identifiant et son nouvel état local. Lors de la réception d'une correction, p_i peut choisir soit de l'accepter soit de la refuser. Il l'accepte si la correction ne lui fait pas perdre d'information ($\forall k, \mathbf{clock}_i[k] \leq cl_j[k]$) et si elle peut lui en faire gagner ($\exists k, \mathbf{clock}_i[k] < cl_j[k]$) ou si le processus qui a envoyé l'information est plus à même que lui de devenir leader finalement ($j \leq \mathbf{leader}_i$).

Proposition 3.6 (Cohérence d'écritures). *Toutes les histoires admises par l'algorithme 3.9 pour un ADT T sont dans $UC(T)$.*

Démonstration. Soient $T \in \mathcal{T}$ et H une histoire autorisée par l'algorithme 3.9. Si H possède un nombre infini d'écritures ou un nombre fini de lectures, $H \in UC(T)$. Sinon, le nombre d'écritures effectuées par p_i est fini. Notons le m_i et soit $cl_{max} = [m_0, ..., m_{n-1}]$. De plus, il existe un instant t_1 à partir duquel tous les processus corrects ont reçu tous

```
1  algorithm UC_0(A, B, Z, ζ_0, τ, δ)
2      variable clock_i ∈ ℕ[n] ← [0, ..., 0] ;    // nombre d'écritures connues
3      variable leader_i ∈ ℕ ← i ;                 // identité du leader
4      variable state_i ∈ Z ← ζ_0 ;                // état courant
5      operation apply (α ∈ A) ∈ B
6          variable β ∈ B ← δ(state_i, α) ;        // lecture
7          if α ∈ U_T then
8              FIFO broadcast mUpdate (clock_i[i] + 1, i, α) ;   // écriture
9          end
10         return β ;
11     end
12     on receive mUpdate (t_j ∈ ℕ, j ∈ ℕ, α ∈ A)
13         if clock_i[j] < t_j then            // écriture pas encore connue
14             clock_i[j] ← t_j ;
15             state_i ← τ(state_i, α) ;  // application locale de l'écriture
16             leader_i ← i ;
17             broadcast mCorrect (clock_i, i, state_i) ;
18         end
19     end
20     on receive mCorrect (cl_j ∈ ℕ[n], j ∈ ℕ, s_j ∈ S)
21         if ∀k, clock_i[k] ≤ cl_j[k] ∧ (j ≤ leader_i ∨ ∃k, clock_i[k] < cl_j[k]) then
22             clock_i ← cl_j ;                // acceptation de la correction
23             leader_i ← j ;
24             state_i ← s_j ;
25         end
26     end
27 end
```

FIGURE 3.9 – Algorithme générique $UC_0(T)$: code pour p_i.

les messages `mUpdate`, donc un instant t_2 à partir duquel ils ont tous reçu tous les messages `mCorrect`.

Prouvons par récurrence sur la suite des états de p_i qu'à tout instant, l'état de p_i résulte d'une linéarisation de l'ensemble U_i des écritures correspondant à un message $\texttt{mUpdate}(cl_j, j, \alpha)$ telles que $cl_j \leq \mathbf{clock}_i[j]$, et $\mathbf{clock}_i \leq cl_{max}$. Initialement, $\mathbf{clock}_i = [0, ..., 0]$ et $\mathbf{state}_i = \zeta_0$ donc la propriété est vérifiée.

Supposons que la propriété est vérifiée et que p_i reçoit un message $\texttt{mUpdate}(cl_j, j, \alpha)$. Si $cl_j \leq \mathbf{clock}_i[j]$, l'état reste inchangé donc la propriété également. Sinon, en raison de la réception FIFO, $cl_j = \mathbf{clock}_i[j] + 1$, donc le nouvel élément de U_i en est un maximum et d'après la ligne 15, le nouvel état correspond à la linéarisation précédente à laquelle est concaténée α. De plus, $\mathbf{clock}_i[j] = cl_j \leq m_j$ donc la propriété reste vérifiée.

Supposons maintenant que le message reçu par p_i est $\texttt{mCorrect}(cl_j, j, s_j)$. Après le traitement du message, soit p_i est dans le même état qu'avant la réception, soit il est dans l'état de p_j au moment où il a envoyé le message. Dans tous les cas, la propriété est vérifiée.

De plus, à l'instant t_2, pour tout j, $\mathbf{clock}_i[j] = m_j$. En effet, les horloges des messages `mUpdate` envoyés par p_j sont successives en raison du $\mathbf{clock}_i[i] + 1$ de la ligne 8, de la

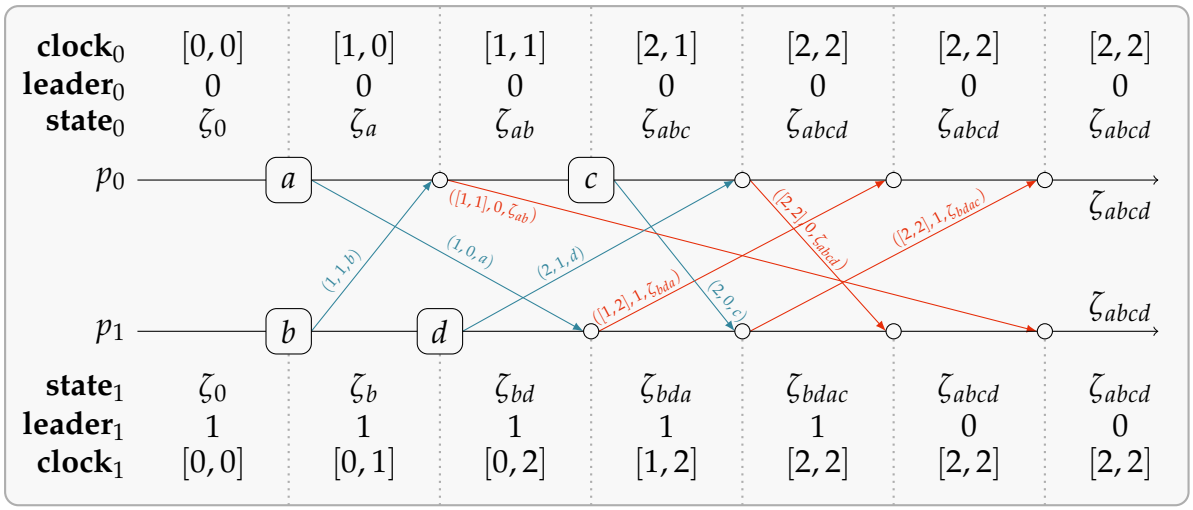


FIGURE 3.10 – Exemple d'exécution de l'algorithme UC_0

ligne 14 et du fait que les messages sont reçus instantanément par leur émetteur. Si la valeur c de $\mathbf{clock}_i[j]$ à t_2 était $c < m_j$, p_j aurait donc envoyé un message $\texttt{mUpdate}(c+1, j, \alpha)$, qui aurait été reçu par p_i avant t_1, et d'après la ligne 14, $c \geq c+1$, ce qui est absurde.

Soit p_i un processus correct. D'après ce qui précède, à l'instant t_2, son horloge vaut cl_{max}. À l'instant où son horloge a pris la valeur cl_{max}, p_i venait de recevoir un message. S'il s'agit d'un message $\texttt{mUpdate}$, il a lui-même diffusé un message $\texttt{mCorrect}$. Sinon, il a reçu un message $\texttt{mCorrect}$ associé à l'horloge cl_{max}. Dans tous les cas un message $\texttt{mCorrect}(cl_{max}, k, \zeta_k)$ a été diffusé. Parmi tous ceux-ci, considérons celui pour lequel k est minimal. Quand p_i reçoit ce message, il a bien $\mathbf{clock}_i \leq cl_{max}$, puisque cl_{max} est la plus grande horloge que p_i peut atteindre durant une exécution. Supposons que $\mathbf{clock}_i = cl_{max}$ et $\mathbf{leader}_i \leq k$. La modification précédente de l'état de p_i ne peut pas être due à la réception d'une correction car par définition de k, aucun message $\texttt{mCorrect}(cl_{max}, \mathbf{leader}_i, \zeta)$ n'a été diffusé. Elle ne peut pas non plus être due à la réception d'une écriture car alors p_i aurait diffusé un message $\texttt{mCorrect}(cl_{max}, i = \mathbf{leader}_i, \zeta)$, ce qui est à nouveau impossible. Ainsi, p_i a accepté la correction de p_k.

Pour tous les processus p_i, le message $\texttt{mCorrect}(cl_{max}, k, \zeta_k)$ pour lequel k est minimal est le même. On en déduit qu'après l'instant t_2, pour tout processus p_i, $\mathbf{clock}_i = \zeta_k$. On sait également qu'il existe une linéarisation de toutes les écritures qui mène dans l'état ζ_k. L'ensemble E' des événements effectués avant l'instant t_2 est fini et $\text{lin}(H[E_H/E']) \cap L(T) \neq \emptyset$. L'histoire H vérifie donc la cohérence d'écritures. □

3.2.3 L'algorithme $UC[k]$

L'algorithme UC_∞ de la figure 3.7 est très efficace en communication. Un unique message est diffusé pour chaque écriture et chaque message ne contient que l'information nécessaire à identifier l'opération et une estampille composée de deux entiers dont la taille ne croît que logarithmiquement avec le nombre de processus et le nombre d'opérations dans l'histoire. Par contre, il est coûteux en mémoire et en temps de calcul local. En effet, la taille de l'espace local en mémoire et le temps nécessaire pour effectuer une lecture croissent linéairement avec le nombre d'opérations.

L'algorithme UC_0 de la figure 3.9 est au contraire efficace en mémoire et en temps

```
1  algorithm UC[k](A, B, Z, ζ0, τ, δ)
2      variable history_i ⊂ ℕ × ℕ × A ← ∅ ;          // événements récents
3      variable vtime_i ∈ ℕ ← 0 ;                    // horloge logique de Lamport
4      variable rstate_i ∈ Z ← ζ0 ;                  // résumé des événements anciens
5      variable rvtime_i ∈ ℕ ← 0 ;                   // limite entre ancien et récent
6      variable rclock_i ∈ ℕ[n] ← [0, ..., 0] ;      // derniers événements anciens
7      variable leader_i ∈ ℕ ← i ;                   // identité du leader
8      variable sent_i ∈ 𝔹 ← false ;                 // empêche l'envoi multiple
9      operation apply (α ∈ A) ∈ B
10         variable q ∈ Z ← rstate_i ;
11         if α ∈ Q_T then                                              // lecture
12             for (t_j, j, α') ∈ history_i triés selon (t_j, j) do q ← τ(q, α')
13         end
14         if α ∈ U_T then FIFO broadcast mUpdate (vtime_i + 1, i, α)   // écriture
15         return δ(q, α) ;
16     end
17     on receive mUpdate (t_j ∈ ℕ, j ∈ ℕ, α ∈ A)
18         vtime_i ← max(vtime_i, t_j) ;   // gestion de l'horloge de Lamport
19         if rclock_i[j] < t_j then                // écriture pas encore connue
20             history_i ← history_i ∪ {(t_j, j, α)} ;
21             variable conflict ∈ 𝔹 ← t_j ≤ rvtime_i ;      // écriture ancienne
22             record(vtime_i) ;
23             if conflict then
24                 broadcast mCorrect (rclock_i, rvtime_i, i, rstate_i) ; sent_i ← true ;
25             end
26         end
27     end
28     on receive mCorrect (cl_j ∈ ℕ[n], t_j ∈ ℕ, j ∈ ℕ, q ∈ Z)
29         if rvtime_i < t_j then record(t_j + k)
30         if ((rclock_i < cl_j) ∨ (rclock_i = cl_j ∧ j < leader_i)) then
               // accepte la correction
31             rclock_i ← cl_j ; rstate_i ← q ; leader_i ← j ; sent_i ← true ;
32         else if (cl_j < rclock_i ∨ (cl_j = rclock_i ∧ leader_i < j)) ∧ j ≠ i ∧ ¬sent_i then
               // refuse la correction et aide p_j
33             broadcast mCorrect (rclock_i, rvtime_i, i, rstate_i) ; sent_i ← true ;
34         end
35     end
36     function record (t ∈ ℕ)
37         rvtime_i ← max(rvtime_i, ((k > 0) ? (k × (⌊t/k⌋ − 1)) : (t))) ;
           // k ≤ t − rvtime_i ≤ 2k
38         for (t_j, j, α) ∈ history_i avec t_j ≤ rvtime_i triés selon (t_j, j) do
               // (t_j, j, α) devient une écriture ancienne
39             rclock_i[j] ← t_j ; history_i ← history_i \ {(t_j, j, α)} ; rstate_i ← τ(rstate_i, α) ;
40             leader_i ← i ; sent_i ← false ;
41         end
42     end
43 end
```

FIGURE 3.11 – Algorithme générique $UC[k](T)$: code pour p_i

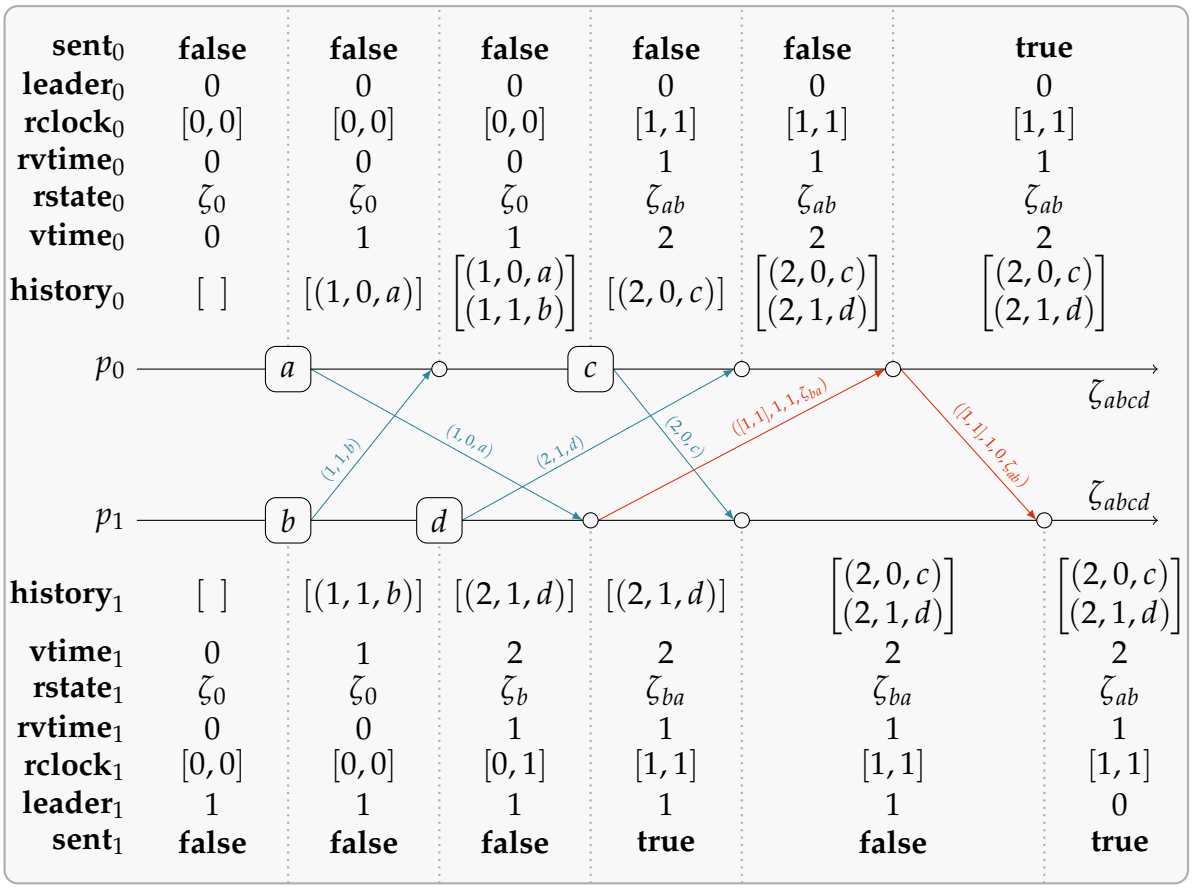


FIGURE 3.12 – Exemple d'exécution de l'algorithme $UC[k]$

de calcul. L'état est immédiatement accessible pour une lecture et la complexité en mémoire est la même que pour la cohérence pipeline, c'est-à-dire un état de l'objet et un vecteur de n entiers dont la taille de chacun croît logarithmiquement avec le nombre d'opérations. Par contre, l'algorithme est quadratique en nombre de messages diffusés par opération. Surtout, la plupart des messages envoyés comportent une horloge vectorielle et un état local. Or l'état local est susceptible d'être très lourd. Par exemple, dans le cas d'une base de données, l'état local est la totalité de la base.

L'algorithme $UC[k]$ sur la figure 3.11 associe les deux méthodes précédentes en gardant leurs meilleurs aspects. Un exemple d'exécution est montré sur la figure 3.12. Les systèmes réels sont rarement complètement asynchrones. En général, l'asynchronie est utilisée comme une abstraction pratique pour modéliser des systèmes dans lesquels les délais de transmission sont en fait bornés (au moins avec une grande probabilité) mais la borne est trop grande pour être utilisée en pratique (on ne peut pas demander à un utilisateur d'attendre plusieurs minutes entre chaque action dans un service interactif), ou inconnue. Cela signifie qu'au bout d'un moment, le préfixe de la liste **history**$_i$ dans l'algorithme UC_∞ sera fixe et les vieux messages pourront être résumés en un état de l'objet. Si un vieux message est reçu plus tard qu'attendu, la stratégie de l'algorithme UC_0 est utilisée pour garantir la cohérence d'écritures. L'ordre total imposé *a priori* par les estampilles peut être modifié si tous les processus appliquent la même modification. Cela se fait en s'échangeant les états locaux. Ainsi, le surcoût en mémoire et en temps de calcul reste borné en fonction d'un paramètre k qui affecte la taille de la liste, et de la bande passante supplémentaire n'est nécessaire que quand des messages sont anormalement retardés, par exemple après une partition. La limite de

l'algorithme $UC[k]$ quand k tend vers l'infini est très semblable à UC_∞ et l'algorithme $UC[0]$ est une version optimisée de UC_0, puisque la liste reste toujours vide, mais des messages de correction ne sont envoyés qu'en cas d'écritures concurrentes.

Le processus p_i gère les variables suivantes.

- $\textbf{history}_i \subset \mathbb{N} \times \mathbb{N} \times \mathsf{A} \leftarrow \emptyset$: la liste des écritures « récentes », reçues mais pas encore appliquées et de taille $\mathcal{O}(n \times k)$.
- $\textbf{vtime}_i \in \mathbb{N}$: l'horloge de Lamport utilisée pour estampiller les écritures.
- $\textbf{rstate}_i \in \mathsf{Z}$: l'état qui résume l'ensemble des écritures « anciennes » qui ont été reçues mais ne font plus partie de $\textbf{history}_i$.
- $\textbf{rvtime}_i \in \mathbb{N}$: la date limite entre les écritures récentes et anciennes. Une écriture est dite « ancienne » si, et seulement si, l'horloge de Lamport de son estampille est inférieure à $\textbf{rvtime}_i$. Sinon elle est « récente ».
- $\textbf{rclock}_i \in \mathbb{N}[n]$: l'horloge vectorielle associée à $\textbf{rstate}_i$, utilisée pour comparer les états. Le nombre $\textbf{rclock}_i[j]$ représente la plus grande horloge linéaire des écritures faites par p_j et appliquées pour construire $\textbf{rstate}_i$.
- $\textbf{leader}_i \in \mathbb{N}$: l'identifiant du dernier processus dont p_i a accepté une correction.
- $\textbf{sent}_i \in \mathbb{B}$: un booléen vrai si et seulement si l'état courant a déjà été diffusé, utilisé pour empêcher les diffusions multiples du même message.

L'état courant d'un processus est obtenu en appliquant toutes les écritures contenues dans $\textbf{history}_i$ à l'état $\textbf{rstate}_i$ dans l'ordre lexicographique des estampilles.

Pour exécuter une écriture, p_i diffuse de manière FIFO un message composé du symbole de l'opération et d'une estampille (t_i, i) créée de façon similaire à l'algorithme UC_∞. Quand un processus reçoit un message $\texttt{mUpdate}(t_j, j, \alpha)$, il essaye de l'insérer dans sa liste d'écritures $\textbf{history}_i$. Il appelle ensuite la méthode $\textbf{record}(t)$ qui s'assure que $\textbf{vtime}_i - \textbf{rvtime}_i < 2k$ et applique les écritures dont l'estampille est inférieure à la nouvelle valeur de $\textbf{rvtime}_i$. Un conflit est détecté si l'horloge du nouveau message t_j est plus petite que celle de l'état $rvtime_i$, ce qui signifie que l'écriture aurait déjà dû être appliquée à $\textbf{rstate}_i$ — et l'a potentiellement été par un autre processus. Pour résoudre le conflit, p_i applique l'écriture localement à $\textbf{rstate}_i$ puis informe les autres processus du conflit en envoyant son nouvel état. Comme dans l'algorithme UC_0, l'ordre de linéarisation reste correct grâce à la réception FIFO sur les messages `mUpdate`.

Quand p_i reçoit un message $\texttt{mCorrect}(cl_j, t_j, j, q)$ de p_j, il sait que p_j a observé un conflit et changé son ordre de linéarisation. Il peut choisir d'accepter ou de refuser la correction, comme dans l'algorithme UC_0, en fonction des estampilles de son état et de la correction, $\textbf{rvtime}_i$ et t_j.

- Si $t_j < \textbf{rvtime}_i$, l'état de la correction est très vieux. Ce cas peut se produire si le message de correction a été reçu avec beaucoup de retard. S'il acceptait la correction, p_i perdrait l'information des écritures contenues entre t_j et $\textbf{rvtime}_i$ qu'il a déjà intégrées à $\textbf{rstate}_i$. Il diffuse à son tour un message de correction pour aider p_j à rattraper son retard.
- Si $t_j > \textbf{rvtime}_i$, p_i est en retard par rapport à p_j. Il avance alors sa propre horloge à t_j pour être dans l'un des cas suivants.

- Si $t_j = \mathbf{rvtime}_i$ et $cl_j = \mathbf{rclock}_i$, les deux états ont été produits en prenant en compte les mêmes écritures mais potentiellement dans un ordre différent. L'arbitrage est alors effectué en comparant l'identifiant des processus, et en utilisant une variable $\mathbf{leader}_i$ comme dans l'algorithme UC_0. Si p_i décide de garder son propre état, il le diffuse pour que les autres processus puissent faire le même choix. Grâce à la variable $\mathbf{send}_i$, un état ne peut être envoyé qu'une seule fois.

- Si $t_j = \mathbf{rvtime}_i$ et $cl_j > \mathbf{rclock}_i$, il existe des messages en transit à destination de p_i, qui vont produire un nouveau conflit dans le futur. Pour éviter ces futurs conflits, p_i accepte la correction.

- Si $t_j = \mathbf{rvtime}_i$ et $cl_j < \mathbf{rclock}_i$, de même, p_j produira un nouveau conflit dans le futur, donc p_i diffuse son état pour annuler les erreurs de p_j.

- Si $t_j = \mathbf{rvtime}_i$ et les deux horloges ne sont pas comparables, choisir un état risque d'entraîner une perte d'information. De plus, on sait que les deux processus auront un nouveau conflit dans le futur, donc ils pourront se mettre d'accord plus tard. La correction est donc simplement ignorée.

Outre l'absence d'attente, ce qui garantit la tolérance aux défaillances dans cet algorithme est que chaque processus garde toujours un état acceptable pour un préfixe de l'histoire. La communication supplémentaire ne sert qu'à gérer la convergence. Or les processus corrects ont tout le temps pour se mettre d'accord (puisqu'ils sont corrects), et les processus fautifs ne peuvent pas effectuer une infinité de lectures, donc ils ne peuvent pas empêcher la convergence.

3.3 Étude de l'algorithme $UC[k]$

3.3.1 Correction

Dans cette partie, nous démontrons que l'algorithme de la figure 3.11 implémente la cohérence d'écritures, c'est-à-dire que toutes les histoires qu'il admet vérifient la cohérence d'écritures. Pour cela, nous prouvons trois lemmes intermédiaires. Le lemme 3.7 assure, entre autres propriétés à propos de l'état global du système, que l'état local virtuel d'un processus est toujours le résultat d'une linéarisation des écritures pour lesquelles il a reçu le message `mUpdate`. Le lemme 3.8 affirme que, si tous les processus arrêtent d'écrire alors, au bout d'un moment, plus aucun message ne sera envoyé pour maintenir la cohérence et le lemme 3.9 prétend qu'alors, tous les processus auront convergé vers un état commun. Finalement, la proposition 3.10 prouve que l'algorithme $UC[k]$ est correct.

Nous considérons un type de données abstrait $T = (\mathsf{A}, \mathsf{B}, \mathsf{Z}, \zeta_0, \tau, \delta)$ et une histoire concurrente $H = (\Sigma, E, \Lambda, \mapsto, \nearrow)$ formée de n séquences parallèles d'événements correspondant aux n processus de l'algorithme. Lors de sa k^{e} écriture, notée u_i^k, p_i diffuse un message $\texttt{mUpdate}(v_i^k, i, \alpha)$. Soit M_{update} l'ensemble des triplets $(v_i^k, i, \alpha) \in \mathbb{N}^2 \times \mathsf{A}$ tels qu'un message $\texttt{mUpdate}(v_i^k, i, \alpha)$ a été diffusé par le processus p_i.

Nous utilisons la notation en exposant pour indiquer la valeur prise par une variable à un instant t. Par exemple, $\mathbf{rstate}_i^t$ est la valeur de la variable $\mathbf{rstate}_i$ du processus p_i à l'instant t. Nous étendons cette notation à deux autres valeurs qui peuvent être calculées à partir des variables. L'état virtuel d'un processus à l'instant t, $\zeta_i^t \in \mathsf{Z}$ est l'état dans lequel sont faites les lectures, correspondant au contenu de la variable

q après la ligne 12. Le vecteur $\mathbf{clock}_i^t$ représente l'estampille de la plus récente écriture connue par p_i de chaque processus :

$$\forall j, \mathbf{clock}_i^t[j] = max(\{\mathbf{rclock}_i^t[j]\} \cup \{t_j : \exists(t_j, j, \alpha) \in history_i^t\})$$

Lemme 3.7 (Validité). *Soient p_i un processus et $t \in \mathbb{R}$ un instant tel qu'un processus vient de terminer l'exécution d'une ligne à l'instant t.*

1. *L'horloge $\boldsymbol{rclock}_i$ de p_i est croissante :*

$$\forall j, \forall t' \leq t, \boldsymbol{rclock}_i^{t'}[j] \leq \boldsymbol{rclock}_i^t[j]$$

2. *L'état enregistré $\boldsymbol{rstate}_i$ de p_i à l'instant t résulte d'une linéarisation des écritures associées à un message $\texttt{mUpdate}(v_i^k, i, \alpha)$ avec $v_i^k \leq \boldsymbol{rclock}_i^t[j]$:*

$$\exists \alpha^1 ... \alpha^l \in lin(H[\{u_j^k : v_j^k \leq \boldsymbol{rclock}_i^{t'}[j]\}/\emptyset]), \tau(\tau(...\tau(\zeta_0, \alpha^1), ...), \alpha^k) = \boldsymbol{rstate}_i^{t'}$$

3. *La variable $\boldsymbol{history}_i$ contient les triplets correspondant aux écritures dont le message $\texttt{mUpdate}(v_i^k, i, \alpha)$ a été reçu par p_i mais qui n'ont pas été enregistrées dans $\boldsymbol{rstate}_i$:*

$$\boldsymbol{history}_i^{t'} = \{(v, j, \alpha) \in M_{update} : \boldsymbol{rclock}_i^{t'}[j] < v \leq \boldsymbol{clock}_i^{t'}[j]\}$$

Démonstration. Nous procédons par induction sur la suite des instants t où un processus vient de terminer l'exécution d'une ligne. Initialement, pour tout processus p_i, $\mathbf{rstate}_i^0 = \zeta_0$, $\mathbf{rclock}_i^0 = [0, ..., 0]$ et $\mathbf{history}_i = \emptyset$ donc les trois propriétés sont vérifiées. Supposons les trois propriétés vraies à l'instant t^- au début de l'exécution d'une ligne par p_i et montrons qu'elles sont encore vraies à l'instant t où p_i termine l'exécution de la ligne.

Ligne 20 : les variables $\mathbf{rclock}_i$ et $\mathbf{rstate}_i$ ne sont pas modifiées donc (1) et (2) restent vraies. Intéressons-nous à (3). On a $\mathbf{history}_i^t = \mathbf{history}_i^{t^-} \cup \{(t_j, j, \alpha)\}$;

Soit $(v', j', \alpha') \in M_{update}$ tel que $\mathbf{rclock}_i^t[j'] < v' \leq \mathbf{clock}_i^t[j']$. Puisque $\mathbf{rclock}_i^t[j'] < \mathbf{clock}_i^t[j']$, il existe un triplet $(\mathbf{clock}_i^t[j'], j', \mathsf{A}'') \in \mathbf{history}_i^t$ par définition de $\mathbf{clock}_i^t$. Ce triplet a été inséré à la ligne 20 donc le message correspondant a été reçu par p_i avant l'instant t. Comme la réception est FIFO, le message $\texttt{mUpdate}(v', j', \alpha)$ a également été reçu par p_i avant t. Ainsi, si $(v', j', \alpha') \notin \mathbf{history}_i^t$, soit il n'a pas jamais été inséré dans $\mathbf{history}_i$, auquel cas la condition de la ligne 19 était fausse au moment de la réception du message ($v' \leq \mathbf{rclock}_i[j]$), soit il a été supprimé depuis à la ligne 39 juste après l'assignation $\mathbf{rclock}_i[j'] \leftarrow v'$. Dans les deux cas, il y aurait eu un instant $t' \leq t$ tel que $v' \leq \mathbf{rclock}_i^{t'}[j]$. D'après l'hypothèse d'induction, on a $\mathbf{rclock}_i^{t'}[j] \leq \mathbf{rclock}_i^{t^-}[j]$. On a donc $v' \leq \mathbf{rclock}_i^{t^-}[j] = \mathbf{rclock}_i^t[j]$, ce qui est en contradiction avec l'hypothèse. On en déduit $(v, j, \alpha) \in \mathbf{history}_i^t$.

Réciproquement, soit $(v', j', \alpha') \in \mathbf{history}_i^t$. Le triplet a été ajouté à la ligne 20 juste après la réception du message $\texttt{mUpdate}(v', j', \alpha')$, donc $(v', j', \alpha') \in M_{update}$. Si $(v', j', \alpha') \in \mathbf{history}_i^{t^-}$, alors $\mathbf{rclock}_i^t[j'] = \mathbf{rclock}_i^{t^-}[j'] < v' \leq \mathbf{clock}_i^{t^-}[j'] \leq \mathbf{clock}_i^t[j']$ par induction. Sinon, $(v', j', \alpha') = (t_j, j, \alpha)$. $\mathbf{rclock}_i^t[j'] = \mathbf{rclock}_i^{t^-}[j'] < v'$ d'après la condition de la ligne 19 et $v' \leq \mathbf{clock}_i^t[j']$ par définition de $\mathbf{clock}_i^t$.

Ligne 31 : Si la ligne s'exécute, la condition de la ligne 30 est vraie, donc pour tout j, $\mathbf{clock}_i^t[j] = cl_j > \mathbf{clock}_i^{t^-}[j]$ ce qui nous donne la propriété (1).

Pour la propriété (2), remarquons que le message $\texttt{mCorrect}(cl_j, t_j, j, q)$ que vient de recevoir p_i a été envoyé par p_j à un instant $t' < t^-$. Que l'envoi ait eu lieu à la ligne 24 ou 33, on a $\mathbf{rclock}_i^t = cl = \mathbf{rclock}_j^{t'}$ et $\mathbf{rstate}_i^t = q = \mathbf{rstate}_j^{t'}$. Par hypothèse d'induction, p_j vérifiait la propriété (2) à t', donc p_i la vérifie à t.

Intéressons-nous à la propriété (3). Soit $(v', j', \alpha') \in M_{update}$ tel que $\mathbf{rclock}_i^t[j'] < v' \leq \mathbf{clock}_i^t[j']$. Puisque $\mathbf{rclock}_i^t[j'] < \mathbf{clock}_i^t[j']$, il existe un triplet $(\mathbf{clock}_i^t[j'], j', \mathsf{A}'') \in history_i^t = history_i^{t^-}$ par définition de $\mathbf{clock}_i^t$, donc $\mathbf{clock}_i^t[j'] = \mathbf{clock}_i^{t^-}[j']$. On a $\mathbf{rclock}_i^{t^-}[j'] \leq \mathbf{rclock}_i^t[j'] < v' \leq \mathbf{clock}_i^t[j'] = \mathbf{clock}_i^{t^-}[j']$, donc $(v', j', \alpha') \in \mathbf{history}_i^{t^-} = \mathbf{history}_i^t$.

Réciproquement, soit $(v', j', \alpha') \in \mathbf{history}_i^t$. $\mathbf{history}_i^t = \mathbf{history}_i^{t^-}$ donc par l'hypothèse d'induction, $(v', j', \alpha') \in M_{update}$ et $\mathbf{rclock}_i^{t^-}[j'] < v' \leq \mathbf{clock}_i^{t^-}[j'] = \mathbf{clock}_i^t[j']$. Grâce à la ligne 29, on a $\mathbf{rclock}_i^t[j'] = cl_j[j'] \leq t_j \leq \mathbf{rvtime}_i^{t^-} < v'$. On a donc bien $\mathbf{rclock}_i^t[j'] < v' \leq \mathbf{clock}_i^t[j']$.

Ligne 39 : Propriété (1). À l'instant t, on a $\mathbf{rclock}_i^t[j] = t_j > \mathbf{rclock}_i^{t^-}[j]$ d'après la propriété 3 appliquée sur le triplet (t_j, j, α) à l'instant t^-. Par induction, on a donc que pour tout $t' \leq t$, $\forall j, \mathbf{rclock}_i^{t'}[j] \leq \mathbf{rclock}_i^t[j]$.

Propriété (2). À l'instant t, on a donc $\tau(\mathbf{rstate}_i^{t^-}, \alpha') = \mathbf{rstate}_i^{t^+}$. En appliquant l'hypothèse, on a $\tau(\tau(\tau(...\tau(\zeta_0, \alpha^1), ...), \alpha^k), \alpha') = \mathbf{rstate}_i^{t^+}$, où $\alpha^1 \cdots \alpha^k \in \text{lin}\left(H[\{u_j^k : v_j^k \leq \mathbf{rclock}_i^{t^-}[j]\}/\emptyset]\right)$. D'après la propriété de $\mathbf{history}_i$, on sait que l'écriture correspondante au triplet (t_j, j, α) est la plus ancienne écriture de p_j qui n'a pas été intégrée à $\mathbf{rstate}_i$, c'est-à-dire $t_j = v_j^{\mathbf{rclock}_i[j]+1}$. On en déduit que $\alpha^1 \cdots \alpha^k \cdot \alpha' \in \text{lin}\left(H[\{u_j^k : v_j^k \leq \mathbf{rclock}_i^{t^+}[j]\}/\emptyset]\right)$.

Propriété (3). Un triplet (v', j', α') appartient à $\mathbf{history}_i^t$ si et seulement si il appartient à $\mathbf{history}_i^{t^-}$ et il est différent de (t_j, j, α), si et seulement si $(v', j', \alpha') \in M_{update}$ et $\mathbf{rclock}_i^{t^-}[j'] < v' \leq \mathbf{clock}_i^{t^-}[j']$ par l'hypothèse d'induction et $(t_j, j) < (v', j')$ d'après l'ordre imposé par la ligne 38, si et seulement si $(v', j', \alpha') \in M_{update}$ et $\mathbf{rclock}_i^{t^-}[j'] < v' \leq \mathbf{clock}_i^t[j']$, car $\mathbf{clock}_i^t = \mathbf{clock}_i^{t^-}$, ce qui est la propriété (3).

Autres lignes : les variables concernées ne sont pas modifiées.

□

Lemme 3.8 (Passivité)**.** *Supposons que le nombre d'écritures contenues dans H est fini. Notons, pour tout i, m_i le nombre d'écritures effectuées par le processus p_i. Il existe un instant $t_{quiet} \in \mathbb{R}$ tel que, pour tout instant $t' > t_{quiet}$, aucun message n'est en transit dans le système et, pour tout processus p_i, $\boldsymbol{clock}_i^{t_{quiet}} = [v_0^{m_0}, ..., v_{n-1}^{m_{n-1}}]$.*

Démonstration. Comme H contient un nombre fini d'écritures et un message `mUpdate` n'est envoyé qu'à la ligne 14, quand une nouvelle écriture est effectuée, seul un nombre fini de messages `mUpdate` sont envoyés. De plus, un processus ne peut pas diffuser deux messages `mCorrect` avec la même valeur de $\mathbf{rclock}_i$: si la diffusion est faite à la ligne 24, **conflict** est vrai donc à l'appel de `record` précédent la diffusion, la boucle **for** a au moins été exécutée pour le nouveau message reçu ($\mathbf{rvtime}_i$ ne peut que croître à la ligne 37) donc $\mathbf{rclock}_i$ a été incrémentée depuis le dernier message envoyé ; si la diffusion est faite à la ligne 33, $\mathbf{sent}_i$ a été positionnée à **false** lors du précédent appel à `record` juste après une incrémentation de $\mathbf{rclock}_i$. Or $\mathbf{rclock}_i$ est bornée par $[v_0^{m_0}, ..., v_{n-1}^{m_{n-1}}]$ (car ses champs ne peuvent que prendre des valeurs qui sont des estampilles de messages), le nombre de messages `mCorrect` envoyés est également fini.

Comme tous les messages finissent par arriver, il existe un instant t_{quiet} tel que pour tout $t > t_{quiet}$, tous les messages ont été reçus et traités. Remarquons que $\mathbf{clock}_i$ est croissante car $\mathbf{rclock}_i$ l'est et la ligne 39 est le seul endroit où des éléments peuvent être supprimés de $\mathbf{history}_i$, mais la mise à jour de $\mathbf{rclock}_i$ permet de garder $\mathbf{clock}_i$ constante à cette ligne. De plus, $\mathbf{clock}_i[j]$ ne peut prendre que des valeurs d'estampilles de messages `mUpdate` envoyés par p_j donc $\mathbf{clock}_i[j] \leq v_j^{m_j}$ et juste après la réception par p_i du dernier message `mUpdate` envoyé par p_j, $\mathbf{clock}_i[j] \geq v_j^{m_j}$. On en déduit que pour tout $t > t_{quiet}$, $\mathbf{clock}_i^t = [v_0^{m_0}, ..., v_{n-1}^{m_{n-1}}]$. □

Lemme 3.9 (Convergence)**.** *Soient deux processus p_i et p_j et un instant $t \in \mathbb{R}$. On est dans l'un des deux cas suivants : soit $\zeta_i^t = \zeta_j^t$, soit il existe un message en transit à l'instant t.*

Démonstration. Soient deux processus p_i et p_j et un instant $t \in \mathbb{R}$. De nombreux cas sont possibles.

1. *Cas $\boldsymbol{clock}_i^t \neq \boldsymbol{clock}_i^t$.*
   Il existe l tel que $\mathbf{clock}_i^t[l] \neq \mathbf{clock}_j^t[l]$. Supposons $\mathbf{clock}_i^t[l] < \mathbf{clock}_j^t[l]$ (l'autre cas est symétrique). La première fois qu'un processus a écrit localement la valeur $\mathbf{clock}_i^t[l]$, il venait de recevoir un message `mUpdate`$(\mathbf{clock}_i^t[l], l, \alpha)$. Si p_i avait reçu le même message, il aurait $\mathbf{clock}_i^t[l] \geq \mathbf{clock}_j^t[l]$ d'après le lemme 3.7, ce qui n'est pas le cas. On en déduit que ce message est toujours en transit.

2. *Cas $\boldsymbol{clock}_i^t = \boldsymbol{clock}_i^t$ et aucun conflit ne s'est produit avant t.*
   À chaque fois que p_i a reçu un message `mUpdate`$(vt_\alpha, k_\alpha, \alpha)$ à un instant t_{rec}, $vt_\alpha > rvtime_i^{t_{rec}}$. Une fois qu'une écriture a été appliquée dans la fonction `record`, aucune écriture avec une estampille plus petite n'est reçue plus tard. La même chose peut être dite à propos de p_j. Les deux processus ont donc reçu les mêmes écritures et les ont appliquées dans le même ordre pour former leur état virtuel, donc $\zeta_i^t = \zeta_j^t$.

3. *Cas $\boldsymbol{clock}_i^t = \boldsymbol{clock}_i^t$ et au moins un conflit s'est produit, résultant en un message* `mCorrect`$(cl_A, rt_A, k_A, \zeta_A)$ dont l'horloge cl_A n'est dominée par celle d'aucun autre message de correction.
   Si le message n'a pas été reçu par tous les processus, il y a bien un message en

transit dans le système. Considérons le dernier moment t_A où un tel message a été reçu par p_i ou p_j. On peut supposer sans perte de généralité que la réception s'est produite chez p_i.

- Cas $cl_A \not\leq \mathbf{rclock}_i^{t_A}$: après la ligne 29, $\mathbf{rvtime}_i^{t_A} \geq rt_A$, donc s'il existe k tel que $\mathbf{rclock}_i^{t_A}[k] < cl_A[k]$, un message $\texttt{mUpdate}(vt'_\alpha, k, \alpha')$ avec $vt'_\alpha \leq rt_A$ a été reçu par p_j mais pas par p_i, qui est donc toujours en transit.

- Cas $cl_A = rclock_i^{t_A} \wedge j < leader_i^{t_A}$: le processus p_i juste après avoir traité le message d'une part, et le processus p_j lors de l'envoi du message d'autre part, avaient les mêmes valeurs pour **rstate**, **rvtime** et **rclock**. Après t_A, tous les messages reçus par p_i sont des corrections avec une horloge inférieure à $\mathbf{rclock}_i^{t_A}$ qui ne modifient pas l'état des variables locales de p_i, soit des messages `mUpdate`. Pour que $\zeta_i^{t_{\text{quiet}}} \neq \zeta_j^{t_{\text{quiet}}}$, il faudrait qu'un nouveau conflit se soit produit entre t_A et t, ce qui est impossible par définition de t_A.

- Cas $cl_A < \mathbf{rclock}_i^{t_A} \vee (cl_A = \mathbf{rclock}_i^t \wedge \mathbf{leader}_i^{t_A} < j)$: un nouveau message de correction a été envoyé avec l'horloge $\mathbf{rclock}_i^{t_A} > cl_A$, ce qui est impossible par définition de t_A.

- Cas $cl_A = \mathbf{rclock}_i^{t_A} \wedge \mathbf{leader}_i^{t_A} = j$: pour que ce cas se produise, il faudrait que p_i reçoive deux messages `mUpdate` du même processus avec la même horloge, ce qui est impossible.

4. *Autres cas*
On a $\mathbf{clock}_i^t = \mathbf{clock}_i^t$ et deux conflits se sont produits, résultant en deux messages $\texttt{mCorrect}(cl_i, rt_i, k_i, \zeta_i)$ et $\texttt{mCorrect}(cl_j, rt_j, k_j, \zeta_j)$ dont les horloges cl_i et cl_j sont incomparables. Considérons les derniers conflits de ce type avant l'instant t. Il existe deux messages $\texttt{mUpdate}(vt, k, \alpha)$ et $\texttt{mUpdate}(vt', k', \alpha')$ tels que $cl_i[k] < vt \leq cl_j[k] \leq rt_j$ et $cl_j[k'] < vt' \leq cl_i[k'] \leq rt_i$. Supposons que $vt \leq vt'$ (l'autre cas est symétrique) alors $vt \leq rt_i$ et que le processus p_{k_i} a reçu le message $\texttt{mUpdate}(vt, k, \alpha)$ après avoir envoyé $\texttt{mCorrect}(cl_i, rt_i, k_i, \zeta_i)$. Au moment de la réception, soit la condition de la ligne 19 est vraie et il y a un conflit juste après avec une horloge supérieure à cl_i, soit elle est fausse et $\mathbf{rclock}_{k_i}[k]$ a été incrémentée à la ligne 31, ce qui signifie qu'il y a eu un conflit avec une horloge supérieure à cl_i. Dans les deux cas, ce deuxième conflit entre en contradiction avec l'hypothèse que les conflits considérés sont les derniers à avoir eu lieu avant t. On en conclut que p_{k_i} n'a pas reçu le message $\texttt{mUpdate}(vt, k, \alpha)$, qui est donc toujours en transit.

Dans tous les cas, on a pu montrer soit qu'il y avait un message en transit, soit l'état virtuel des deux processus sont égaux, ce qui conclut ce lemme.

□

Proposition 3.10 (Cohérence d'écritures)**.** *Toutes les histoires acceptées par l'algorithme 3.11 vérifient la cohérence d'écritures.*

Démonstration. Si H contient une infinité d'écritures ou un nombre fini de lectures, elle vérifie la cohérence d'écritures par définition. Nous supposons que H contient un nombre fini d'écritures et une infinité de lectures. Un processus p_i effectue m_i écritures. D'après le lemme 3.8, il existe un instant $t_{quiet} \in \mathbb{R}$ tel que, pour tout instant $t > t_{quiet}$, aucun message n'est en transit dans le système et, pour tout processus p_i, $\mathbf{clock}_i^{t_{quiet}} = [v_0^{m_0}, ..., v_{n-1}^{m_{n-1}}]$. Soit $t > t_{quiet}$. D'après le lemme 3.9, soit pour tout couple de processus p_i et p_j, $\zeta_i^t = \zeta_j^t$, soit il existe un message en transit à l'instant t. On sait qu'il n'existe pas de message en transit, dont il existe un état de convergence ζ tel que, pour tout processus p_i, $\zeta_i^t = \zeta$.

Comme le nombre de lectures est infini, il existe un processus correct p_i qui fait une infinité de lectures. On construit une relation d'ordre $\leq$ sur E par, pour tous $e, e' \in E$, $e \leq e'$ si $e \mapsto e'$ ou si e a eu lieu avant t_{quiet} et e' a eu lieu après t_{quiet} ou si e et e' sont des écritures associées aux estampilles (v, j) et (v', j') et l'une des conditions est vérifiée :

- $v \leq rclock_i^{t_{\text{quiet}}}[j]$ et $v' \geq rclock_i^{t_{\text{quiet}}}[j']$, c'est-à-dire e est compris dans $rstate_i^{t_{\text{quiet}}}$ et e' est dans $history_i^{t_{\text{quiet}}}$;
- $v \leq rclock_i^{t_{\text{quiet}}}[j]$, $v' \leq rclock_i^{t_{\text{quiet}}}[j']$ et e apparaît avant e' dans la linéarisation donnée par le lemme 3.7.2 ;
- $v \geq rclock_i^{t_{\text{quiet}}}[j]$, $v' \geq rclock_i^{t_{\text{quiet}}}[j']$ et $(v, j) < (v', j')$.

Soit E' l'ensemble des événements qui se produisent après t_{quiet}. E' ne contient que des lectures pures. On a bien que $E_H \setminus E'$ est fini. On remarque que $\leq$ est un bel ordre sur E qui contient $\mapsto$, donc il existe une linéarisation $l \in \text{lin}\left(H^{\leq}[E_H / E']\right)$.

L'exécution depuis l'état ζ_0 de toutes les écritures d'estampille (v, j) avec $v \leq \mathbf{rclock}_i^{t_{\text{quiet}}}[j]$ selon l'ordre $\leq$ mène à l'état $\mathbf{rstate}_i^{t_{\text{quiet}}}$ et l'exécution depuis l'état $\mathbf{rstate}_i^{t_{\text{quiet}}}$ des écritures d'estampille (v, j) avec $\mathbf{rclock}_i^{t_{\text{quiet}}}[j] < v \leq \mathbf{clock}_i^{t_{\text{quiet}}}[j] = v_j^{m_j}$ selon l'ordre $\leq$ mène à l'état $\zeta_i^{t_{quiet}} = \zeta$. On en déduit que l'exécution de toutes les écritures de H selon l'ordre $\leq$ mène de l'état ζ_0 à l'état ζ. Or l'état ζ est celui dans lequel sont fait toutes les lectures de p_i faites après t_{quiet}, donc tous les événements de E'. On en déduit que $l \in L(T)$.

Finalement $\text{lin}\left(H[E_H / E']\right) \cap L(T) \neq \emptyset$ donc $H \in UC(T)$.

□

3.3.2 Complexité

Dans cette partie, on s'intéresse à l'évaluation de la complexité de l'algorithme $UC[k]$. On considère un système formé de n processus exécutant un nombre fini m d'écritures et exécutant l'algorithme 3.11 pour un k fixé. La complexité de l'algorithme $UC[k]$ est difficile à calculer précisément car elle dépend énormément de l'ordre de réception des messages. La complexité en temps de calcul (appels à la fonction de transition τ lors des lectures) et en espace est directement proportionnelle à la longueur de la

liste $history_i$. Concernant le coût de communication, la taille d'un état est potentiellement beaucoup plus grande que celle d'une opération. De plus, le nombre de messages `mUpdate` envoyés est égal au nombre d'écritures effectuées. Le paramètre important pour mesurer la complexité en communication est donc le nombre de messages `mCorrect` envoyés. Nous introduisons les quatre grandeurs suivantes.

- H_i^t est le nombre de triplets présents dans la variable locale $history_i$ du processus p_i à l'instant t.
- $H = \max_{i\in\{0,\dots,n-1\},0\leq t\leq t_{\text{quiet}}}(H_i^t)$ est la taille maximale de la liste observée sur n'importe quel processus pendant toute l'histoire (toutes les variables restent constantes après t_{quiet}, défini comme dans la proposition 3.10).
- B_i^t est le nombre total de messages `mCorrect` diffusés par le processus p_i entre son démarrage et l'instant t.
- $B = \sum_{i=0}^{n-1} B_i^{t_{\text{quiet}}}$ est le nombre total de messages `mCorrect` diffusés pendant toute l'histoire.

L'algorithme UC_∞ représente le pire cas pour H ($H = m$) et le meilleur cas pour B ($B = 0$) et l'algorithme UC_0 représente le pire cas pour B ($B = m \times n$) et le meilleur cas pour H ($H = 0$). On peut aussi déduire de la ligne 37 de l'algorithme que $H \leq 2 \times n \times k$. On a donc $0 \leq H \leq \min(m, 2 \times n \times k)$ et $0 \leq B \leq m \times n$. Ces bornes ne sont pas suffisantes pour évaluer l'influence de k sur H et B. Nous évaluons cette influence par simulation.

Paramètres généraux

Le type utilisé dans les expériences n'a pas une grande importance puisque l'algorithme est générique. Nous avons utilisé un type basé sur la multiplication de matrices à coefficients rationnels de taille 3×3. L'intérêt est que la taille de l'état est constante (pour un objet dont la taille est linéaire par rapport à m, l'algorithme UC_∞ n'implique pas de surcharge significative) sans être trivialement convergent puisque la multiplication de matrices n'est pas commutative. De plus, toutes les écritures sont nécessaires pour connaître l'état de convergence, à la différence du registre, où une lecture ne dépend que de la dernière écriture. Les états de l'objet considéré sont les matrice réelles carrées de dimension 3. L'état initial est la matrice identité $I_3 = [[1,0,0][0,1,0][0,0,1]]$. L'objet possède une écriture pure pour chaque matrice, dont l'effet est la multiplication à droite de l'état courant par cette matrice. La seule lecture retourne l'état courant. Plus précisément, l'objet est défini par l'ADT :

$$\Big(\mathsf{A} = \{mult(M) : M \in \mathbb{Q}^{3\times 3}\} \cup \{read\}, \mathsf{B} = \{\bot\} \cup \mathbb{Q}^{3\times 3}, \mathsf{Z} = \mathbb{Q}^{3\times 3}, \zeta_0 = I_3, \tau, \delta\Big)$$

$$\tau : \left\{ \begin{array}{ccc} \mathsf{Z}\times\mathsf{A} & \mapsto & \mathsf{Z} \\ (q, mult(M)) & \to & q \times M \\ (q, read) & \to & q \end{array} \right. \qquad \delta : \left\{ \begin{array}{ccc} \mathsf{Z}\times\mathsf{A} & \mapsto & \mathsf{B} \\ (q, mult(M)) & \to & \bot \\ (q, read) & \to & q \end{array} \right.$$

Dans toutes nos expériences, nous avons utilisé une simulation de Monte-Carlo pour évaluer les paramètres H et B. Nous avons simulé un réseau de $n = 10$ processus. Nous modélisons le temps entre deux écritures d'un même processus et la latence

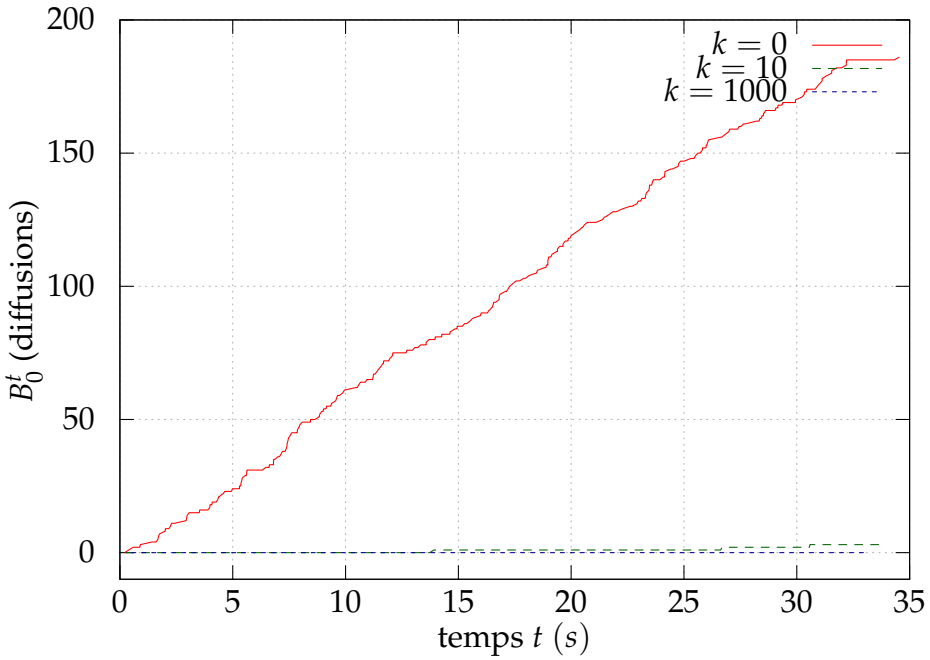
du réseau par deux variables aléatoires suivant chacune une loi exponentielle de paramètre μ et λ respectivement. Seul le rapport $\frac{\lambda}{\mu}$ est important, puisque multiplier λ et μ par une même constante revient à changer l'échelle de temps. Nous avons donc fixé μ à 1Hz et fait varier la valeur de λ. Le rapport $\frac{\lambda}{\mu}$ représente la capacité du réseau à réagir aux écritures : il peut être vu comme l'inverse du nombre moyen d'écritures faites par un processus entre l'émission et la réception d'un message. Plus $\frac{\lambda}{\mu}$ est grand, plus la diffusion se rapproche d'une diffusion atomique instantanée. Ces résultats intègrent une légère optimisation liée à l'implémentation de la réception FIFO : quand un processus peut recevoir plusieurs messages simultanément, il ne diffusera qu'un seul message contenant les corrections pour l'ensemble des messages reçus. On peut aisément adapter la preuve de la proposition 3.10 pour s'assurer que cette optimisation ne change pas la correction de l'algorithme.

Comportement typique

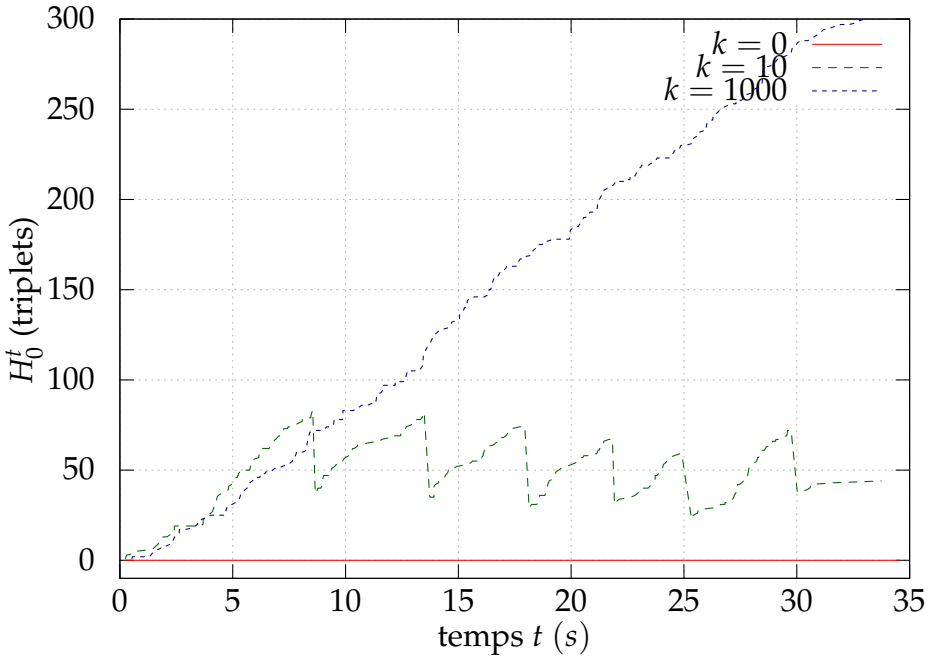
Nous illustrons tout d'abord le comportement de l'algorithme en observant un processus représentatif pour trois valeurs de k : 0, 10 et 1000. Dans la première expérience, le nombre total d'écritures est $m = 300$ et $\frac{\lambda}{\mu} = 1$. La figure 3.13 représente l'évolution du surcoût en mémoire, H_0^t (figure 3.13b) et en communication B_0^t (figure 3.13a) du processus p_0 au cours du temps. Pour $k = 1000$, la liste **history**$_i$ peut stocker toutes les écritures (car $1000 > 300$), donc l'algorithme se comporte comme UC_∞ : aucun conflit n'est observé ($B_0^t = 0$) et H_0^t est croissant pendant toute l'expérience jusqu'à $H = 300$. Au contraire, pour $k = 0$, $H = 0$ mais de nombreux conflits sont observés (environ 180 pour p_0). Ce nombre reste largement inférieur aux $m = 300$ conflits par processus générés (de façon déterministe) par l'algorithme UC_0. Dans $UC[0]$, les messages qui arrivent dans un ordre compatible avec l'ordre des estampilles ne produisent pas de conflit. C'est en particulier le cas pour toutes les messages envoyés par p_i à lui-même, qu'il reçoit immédiatement. Le cas $k = 10$ illustre l'intérêt de l'algorithme par rapport à ces deux limites. Malgré un nombre de conflits très faible (égal à 3 dans cette simulation), le surcoût en mémoire est borné et reste compris entre 30 et 80. Cette limite est également bien inférieure à la borne théorique $2kn = 200$. Cela est dû aux horloges de Lamport, pour lesquelles toutes les valeurs ne sont pas forcément atteintes. Par exemple, dans une exécution sans aucune écriture concurrente, chaque écriture aurait une estampille différente et $H = 2k$.

Comportement en présence de partition

Dans des systèmes répartis réels comme Internet et les centres de données à grande échelle, le délai d'acheminement moyen n'est que rarement constant. En fait, on observe plutôt une alternance de périodes favorables, où les messages sont très rapides, et des périodes instables, comme des partitions du réseau. Dans la deuxième expérience, nous étudions le comportement de l'algorithme en présence de partitionnement. La figure 3.14 présente les ressources utilisées par p_0. Les $n = 10$ processus effectuent un total de $m = 10000$ écritures et $k = 10$. La latence moyenne évolue au cours du temps. De $0s$ à $200s$, de $400s$ à $600s$ et de $800s$ à la fin, le réseau fonctionne normalement et $\frac{\lambda}{\mu} = 1$. Le réseau est partitionné entre $200s$ et $400s$ et entre $600s$ et $800s$. Pendant une partition, tous les messages sont retardés jusqu'à la fin de la partition. Comme précédemment, la figure 3.14 représente l'évolution de H_0^t et B_0^t au cours du temps. En



(a) Nombre de diffusions de corrections.



(b) Nombre d'opérations en mémoire.

FIGURE 3.13 – Une exécution pour le processus p_0, avec $k = 0$, $k = 10$ et $k = 1000$.

dehors de partitions, le comportement observé est le même que dans la première expérience : H_0^t admet des variations très rapides, mais sa valeur reste bornée. Le nombre de conflits est relativement faible (autour de 3 conflits pour 1000 messages `mUpdate` reçus). Pendant les partitions, H_0^t décrit des dents de scie variant de $k = 10$ à $2k = 20$ et aucun conflit n'est observé. Du point de vue de p_0, tout se passe comme s'il était le seul processus correct dans le réseau. Les seules notifications d'écriture qu'il reçoit sont celles qu'il a générées avec des estampilles successives. La taille de $\textbf{history}_0$ croît donc selon l'apparition des écritures de p_0, et décroît brusquement de k à chaque fois que

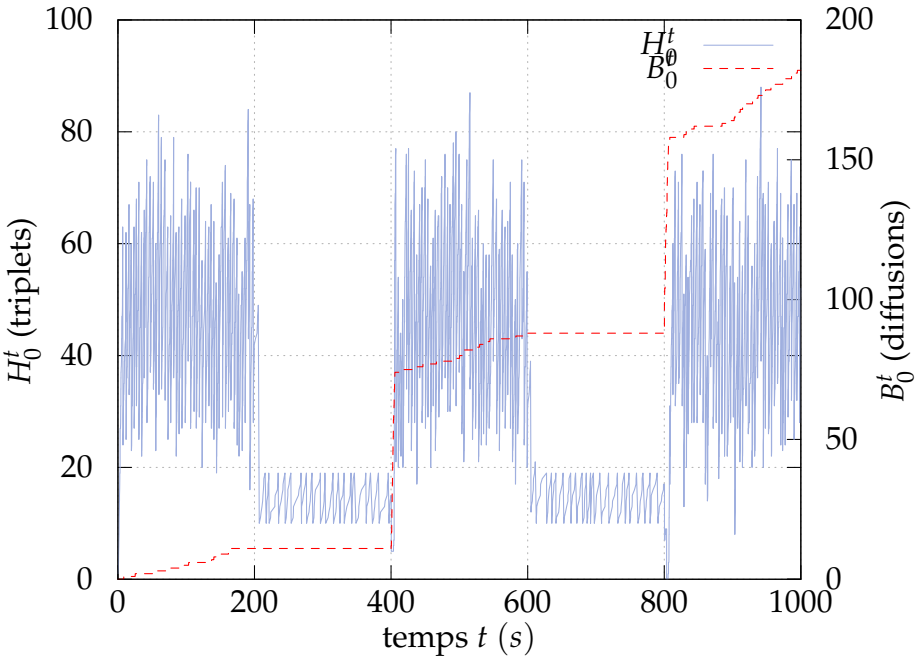


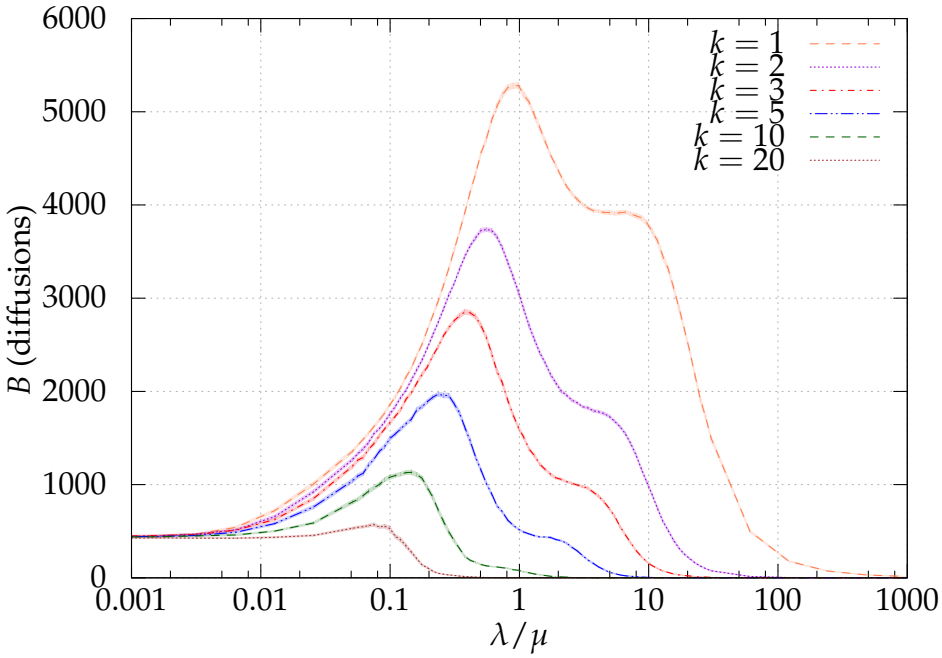
FIGURE 3.14 – Une exécution pour un processus sujet au partitionnement, avec $k = 10$. Les partitions se produisent entre 200s et 400s et entre 600s et 800s.

sa taille atteint $2k$. Comme p_0 reçoit les messages dans l'ordre des estampilles, il n'y a jamais de conflit. À la résolution de la partition, de nombreux conflits sont détectés (environ 70), et le système se rétablit très rapidement (une dizaine de secondes).

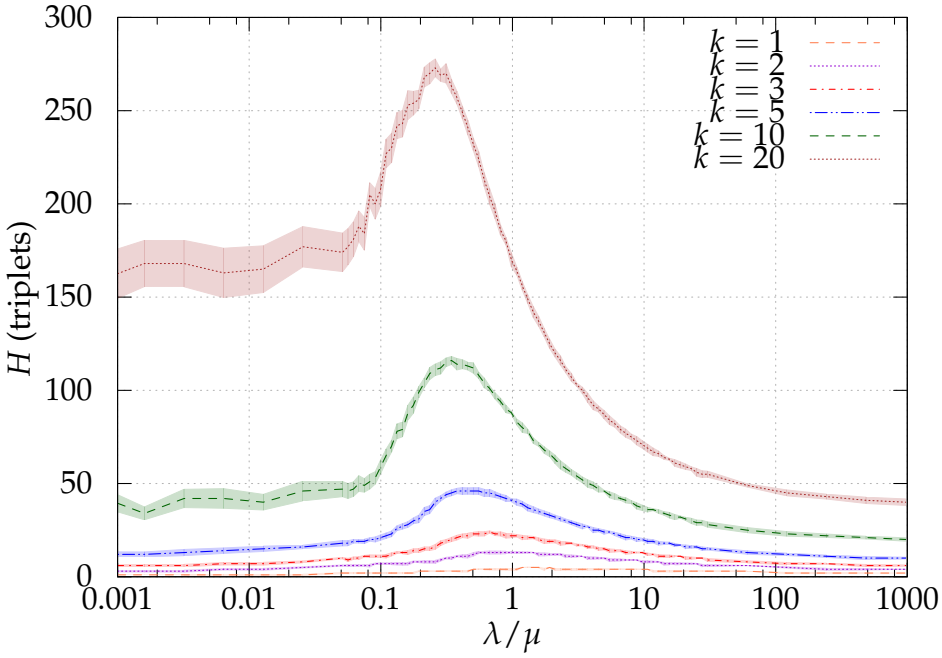
Étude de l'influence de k

Dans la troisième expérience (figures 3.15 et 3.16), on étudie le comportement de l'algorithme en fonction de $\frac{\lambda}{\mu}$ en faisant varier λ entre 0.001Hz et 1000Hz, en le multipliant par 2 à chaque pas près des extrémités de l'intervalle et par 1.1 entre 0.5Hz et 30Hz. Chaque expérience a été conduite cent fois : l'intervalle de confiance à 95% est représenté par une zone colorée autour de la valeur moyenne. Le fait que les intervalles de confiance ne s'intersectent jamais signifie que le nombre d'expériences menées suffit pour comprendre le fonctionnement de l'algorithme. Dans l'expérience, $n = 10$, $m = 1000$ et k prend les valeurs 1, 2, 3, 5, 10 et 20. Les valeurs de B sont représentées sur la figure 3.15a et celles de H sont représentées sur la figure 3.15b.

Les courbes pour les différentes valeurs de k ont la même allure. Quand $\frac{\lambda}{\mu}$ est très petit (inférieur à 0.01Hz), le nombre de corrections diffusées est relativement faible ($B \simeq 400$). Quand $\frac{\lambda}{\mu} = 0.001$Hz, un processus peut effectuer en moyenne 100 écritures avant de recevoir le premier message d'un autre processus. On se retrouve donc dans une situation similaire aux phases de partitionnement de la deuxième expérience : les processus commencent par faire leurs propres écritures localement avant de se synchroniser. Comme tous les messages de correction contiennent au moins toutes les écritures d'un processus, accepter une correction accroît énormément la connaissance du système et peu sont nécessaires. Quand $\frac{\lambda}{\mu}$ croît, le nombre de corrections atteint un maximum puis décroît vers 0. Cette diminution est due à deux facteurs qui s'appliquent à des valeurs différentes de $\frac{\lambda}{\mu}$, ce qui explique le plateau sur les courbes. Premièrement, comme le temps de transmission des messages est réduit, le nombre de



(a) Nombre total de messages de correction diffusés pendant une exécution.



(b) Nombre maximal d'entrées observées dans la mémoire tampon d'un processus pendant une exécution.

FIGURE 3.15 – Surcoût de l'algorithme $UC[k]$ en fonction de $\frac{\lambda}{\mu}$, avec $n = 10$, $m = 1000$ et plusieurs valeurs de k.

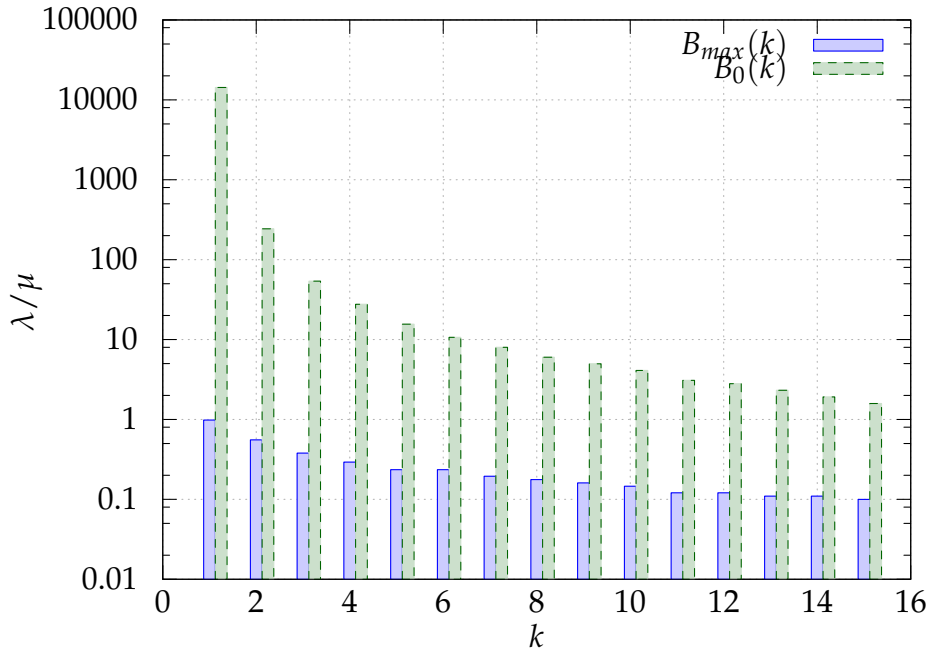


FIGURE 3.16 – Décalage des points d'intérêt de la courbe de $B\left(\frac{\lambda}{\mu}\right)$ en fonction de k. Le pire cas correspond au maximum de la courbe, et le meilleur cas à la plus petite valeur de $\frac{\lambda}{\mu}$ pour laquelle aucune correction n'a été envoyée.

messages qui sont reçus trop tard pour être insérés dans la liste $\mathbf{history}_i$ est réduit. Il s'ensuit une diminution du nombre de conflits. Cet effet s'accompagne d'une augmentation de la taille de la liste des écritures. Deuxièmement, quand les messages sont très rapides, il y a moins de croisements possibles, donc les messages ont tendance à être totalement ordonnés. Les écritures sont alors intégrées dans l'ordre à l'état $\mathbf{rstate}_i$ et moins de conflits se produisent. Durant cette phase, de moins en moins d'écritures sont associées à la même horloge de Lamport, donc H décroît vers $2k$. À la limite où les messages sont transmis instantanément, tous les processus exécutent toutes les opérations dans le même ordre, donc aucun conflit ne se produit et la cohérence séquentielle est vérifiée.

Remarquons qu'augmenter la valeur de k a principalement trois effets : plus k est grand, plus la taille de mémoire utilisée est importante, moins le réseau est sollicité et plus les courbes sont déplacés vers les petites valeurs de $\frac{\lambda}{\mu}$. Le premier effet est très intuitif puisque k est utilisé dans l'algorithme pour limiter la taille de $\mathbf{history}_i$. Pour $\frac{\lambda}{\mu} = 1$, on observe par exemple que $H \simeq 8k$.

Les deux autres effets prouvent que l'utilisation de la mémoire tampon compense un réseau défavorable. Pour étudier l'effet de k sur la position de la courbe, nous définissons deux *points d'intérêt* de la figure 3.15, dont la figure 3.16 montre l'évolution en fonction de k.

- $B_{\max}(k)$ est la valeur de $\frac{\lambda}{\mu}$ pour laquelle le nombre de corrections est maximal
- $B_0(k)$ est la plus petite valeur de $\frac{\lambda}{\mu}$ pour laquelle on observe moins d'un conflit par simulation en moyenne. Cela correspond à un choix optimal de k, celui pour lequel les conflits n'arrivent presque jamais, mais on n'utilise pas de mémoire superflue.

Conclusion

Ce chapitre introduit deux nouveaux critères de cohérence, la cohérence d'écritures et la cohérence d'écritures forte, qui sont respectivement plus forts que la convergence et la convergence forte et plus faibles que la cohérence séquentielle. En pratique, un objet vérifiant la cohérence d'écritures peut être vu comme un objet séquentiellement cohérent, dont l'état peut être corrompu mais capable de se réparer de lui-même après des périodes d'instabilité. Plus le réseau est rapide et la fréquence des écritures sur l'objet est faible, plus les périodes d'instabilité sont brèves. Ce critère convient donc surtout aux systèmes contrôlés par des humains capables de s'adapter à des erreurs temporaires qu'ils peuvent détecter facilement et corriger. En revanche, comme toute lecture est susceptible d'être erronée, il est plus difficile de programmer en utilisant des objets vérifiant la cohérence d'écritures.

Nous avons présenté trois algorithmes génériques pour implémenter la cohérence d'écritures et la cohérence d'écritures forte dans $AS_n[\emptyset]$. Le premier, UC_∞ est très économe en communication puisqu'il se contente de diffuser un message à chaque écriture, mais coûteux en espace mémoire et en temps de calcul puisqu'il conserve toute l'histoire en mémoire et la rejoue à chaque lecture. Au contraire, UC_0 est coûteux en communication mais peu coûteux en espace et en temps de calcul. Le troisième offre un compromis entre les deux premiers pour être économe à la fois en espace et en communication. Il est paramétré par un entier k qui détermine l'espace maximal requis en mémoire. Pour $k = 0$, il s'agit d'une optimisation de UC_0 et plus k est grand, plus son comportement se rapproche de UC_∞. Il existe une valeur de k optimale en fonction du rapport de la vitesse moyenne de transmission des messages sur le réseau et de la fréquence des opérations d'écriture, pour laquelle l'algorithme n'utilise pas plus de bande passante que UC_∞ tout en utilisant le minimum d'espace en mémoire. Le problème est que le calibrage de k est difficile *a priori*. Une piste de recherche serait de permettre à l'algorithme de modifier k à la volée pour s'optimiser en cours d'exécution. La difficulté est de conserver un k identique sur tous les processus.

Une autre piste de recherche concerne les implémentations spécifiques à un objet en particulier. La structure particulière de certains types de données abstraits permet des optimisations qui rendent une implémentation spécifique plus efficace que l'implémentation générique. Cela est par exemple vrai dans le cas de l'ensemble. Dans l'algorithme UC_0, il est nécessaire de conserver l'histoire complète en mémoire car l'état d'un objet de type quelconque peut potentiellement dépendre de toutes les opérations présentes dans l'histoire. Pour l'ensemble, seule la dernière opération d'insertion ou de suppression sur chaque élément a une influence sur l'état lu. On peut donc ne conserver en mémoire que l'ensemble des éléments insérés au moins une fois, associés à l'estampille de leur dernière écriture.

4

La cohérence causale

Sommaire

Préambule

Pour comprendre pourquoi Bob était autant en colère contre Alice, il importe de savoir la relation particulière qu'il entretenait avec son chat. Aussi, l'absence des miaulements de bienvenue lorsqu'il rentra chez lui un soir lui fit un choc qui aggrava son anxiété naturelle. Plus tard dans la soirée, le manque de ronronnements familiers sur ses genoux l'encouragea à organiser une expédition de sauvetage pour retrouver son acolyte à quatre pattes. L'organisation d'une mission d'une telle importance passa par une session de messagerie instantanée en groupe avec ses deux meilleures amies, Alice et Carole. Pour ne pas vexer cette dernière, ils utilisaient l'application développée par Carole qui, une fois l'effet de surprise recherché dissipé, avait été modifiée pour transmettre les messages correctement. L'appel à l'aide n'eut pas l'effet escompté : Alice lui répondit de but en blanc qu'il avait de la chance que son chat soit parti ! Devant autant

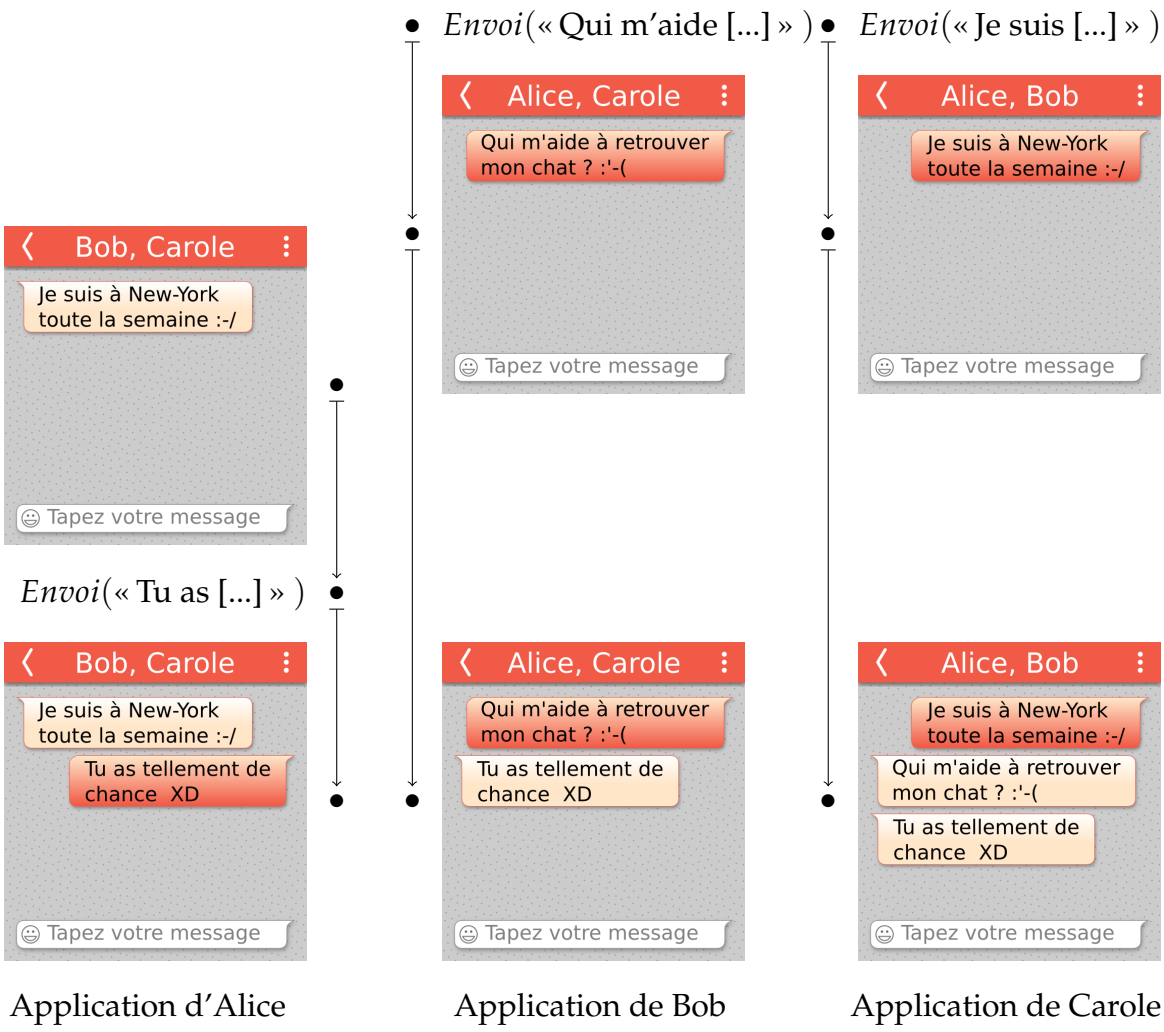


FIGURE 4.1 – Alice parle-t-elle du chat de Bob ou des vacances de Carole ?

de cynisme, il déclama une bordée d'injures et quitta la session. Alice ne comprit pas la réaction de Bob. Après tout, elle n'avait fait que répondre à Carole qui venait de lui apprendre qu'elle était en vacances à New-York pour la semaine, ce qui avait toutes les raisons de la réjouir. Elle ignorait tout des malheurs de son ami d'enfance.

Les deux jeunes femmes continuèrent à discuter quelque temps pour essayer de comprendre ce qui avait mis Bob dans une telle colère. Elles comprirent finalement que Bob n'avait pas reçu le message de Carole, mais une question restait en suspens : si Bob était resté un peu plus longtemps, il aurait sûrement fini par recevoir le message de Carole. Cela aurait pu l'apaiser... si ses messages avaient été réordonnés à la manière de Skype. Bob aurait alors dû reconnaître que le message d'Alice était une réponse à celui de Carole. Dans le cas contraire cela aurait été encore pire : il aurait pu penser que Carole en profitait pour le narguer.

Introduction

L'histoire sur la figure 4.1 résume les étapes de la conversation (ces images ne sont pas issues d'une expérience réelle). Initialement, Bob a envoyé à Alice et Carole le message « Qui m'aide à retrouver mon chat ? :'(». Simultanément, Carole a envoyé le message

« Je suis à New-York toute la semaine :-/ ». Un délai a eu lieu entre l'envoi du message de Bob et sa réception par Alice, si bien qu'Alice a reçu le message de Carole avant celui de Bob. De son point de vue, il n'y a donc aucune ambiguïté sur le sens de sa réponse : « Tu as tellement de chance XD ». Pourtant, Bob reçoit le message d'Alice avant celui de Carole. Il pense donc qu'Alice lui répond directement.

Le problème avec cette histoire est d'ordre temporel : Alice a répondu à la remarque de Carole après l'avoir lue, et elle n'a pu la lire qu'après que Carole l'ait envoyée. L'envoi du message d'Alice étant postérieur à l'envoi de celui de Carole, et il est anormal que Bob reçoive le message d'Alice avant celui de Carole. La relation d'ordre présenté intuitivement ici est généralement appelé *ordre causal*.

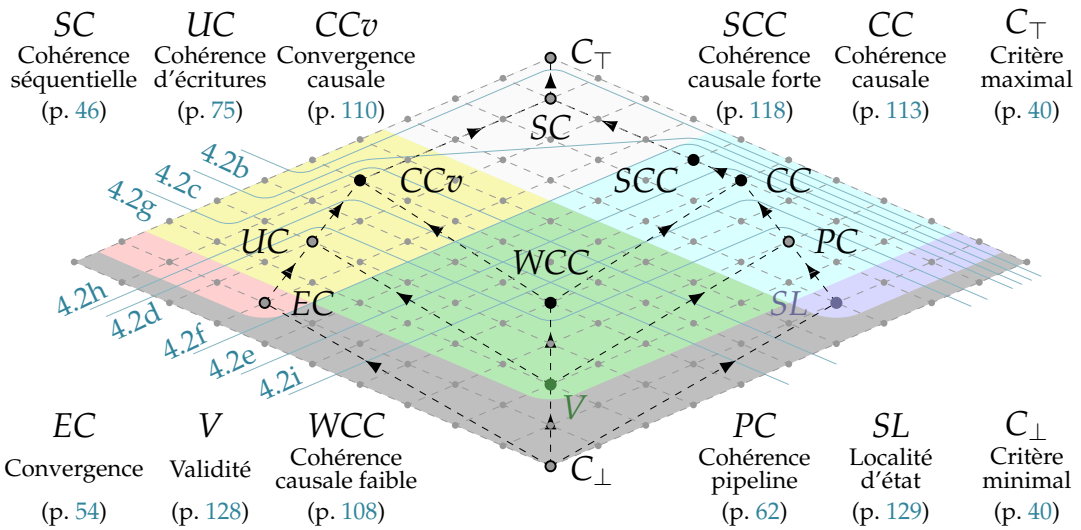
La mémoire causale (voir page 66) est une propriété plus forte que PRAM (l'application de la cohérence pipeline à la mémoire) qui empêche des comportements similaires pour la mémoire. Pour savoir si une histoire vérifie cette propriété, il faut tenter de construire un ordre partiel, appelé *ordre causal*, qui contient l'ordre de processus et une dépendance d'*écriture-lecture* qui encode le fait que toute valeur lue dans un registre doit avoir été écrite précédemment dans le même registre. L'ordre partiel ainsi créé doit être compatible avec, pour chaque processus, une linéarisation contenant les lectures de ce processus et les écritures de tous les processus.

Problème. *Deux problèmes principaux sont soulevés par la définition de la mémoire causale.*

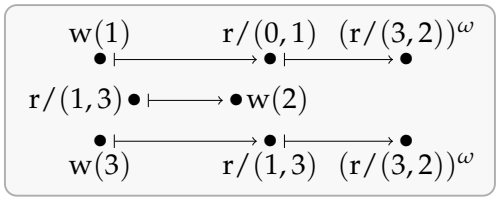
- *Les dépendances entre les lectures et les écritures sont un aspect central dans la définition de la mémoire causale, mais elles sont intimement liées à la spécification séquentielle de la mémoire. Comment étendre la définition de la cohérence causale aux autres types de données abstraits ?*
- *L'ordre causal étant seulement un ordre partiel, des écritures concurrentes peuvent toujours exister. Concrètement, la discussion finale entre Alice et Carole montre que la causalité ne renforce ni la cohérence pipeline ni la cohérence d'écritures, et donc le problème posé dans l'introduction de la thèse se pose de nouveau. Comment la cohérence pipeline et la cohérence d'écritures peuvent-elles être adaptées pour prendre en compte la causalité ?*

L'approche naturelle pour étendre la cohérence causale serait de définir une notion de *dépendance sémantique* applicable à tous les types de données abstraits. Les dépendances sémantiques de la mémoire seraient les dépendances entre les lectures et les écritures, pour les services de messagerie instantanée, la lecture d'un message dépendrait sémantiquement de son envoi, etc. Cette approche semble en faite très dangereuse car la notion de dépendance sémantique est très floue, chaque type de données abstrait ayant ses propres spécificités. Aussi, toute tentative de définition formelle serait forcément discutable. Voici deux exemples qui illustrent les difficultés qui peuvent se rencontrer. Il est bien sûr possible de répondre individuellement à chaque cas particulier en complexifiant la notion de dépendances sémantique, mais qu'en est-il du cas général ?

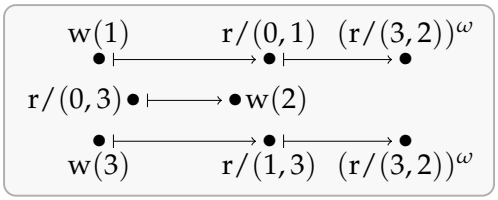
Le compteur. Imaginons le plus simple des compteurs, qu'il est seulement possible d'incrémenter et de lire. La valeur retournée par une lecture ne dépend pas spécifiquement d'une écriture : elle indique seulement le nombre d'écritures dont elle dépend. Comment savoir lesquelles ? Le problème est d'ailleurs également présent pour la mémoire [81] : si une même valeur est écrite plusieurs fois dans un même registre, comment déterminer de quelle écriture dépend une lecture de cette valeur dans ce registre ? Nous verrons dans la partie 4.2.2 que cette question pose une véritable limite de la mémoire causale, à laquelle notre approche permet de répondre.



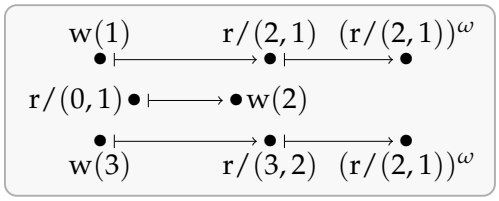
(a) Force relative des différents critères causaux.



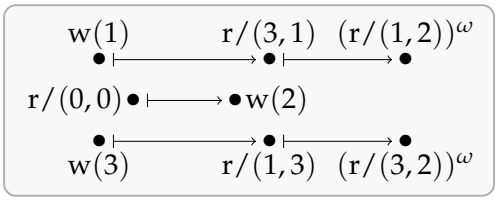
(b) *SC*



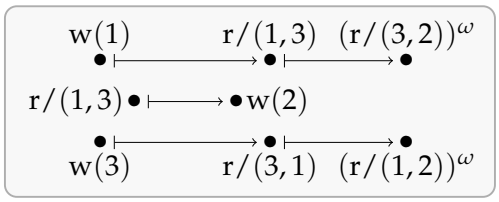
(c) *CCv* et *SCC* mais pas *SC*



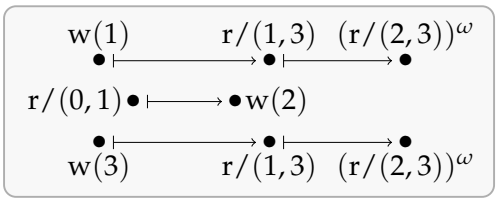
(d) *PC* et *UC* mais pas *WCC*



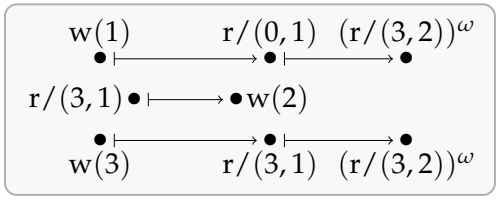
(e) *CC* mais pas *SCC* ni *EC*



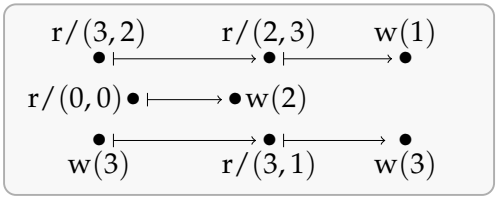
(f) *SCC* mais pas *EC*



(g) *CCv* mais pas *PC*



(h) *WCC* et *UC* mais pas *CCv*



(i) *WCC* et *PC* mais pas *CC*

FIGURE 4.2 – Trois processus partagent une instance de flux fenêtrés ($\mathcal{W}_2$). Dans toutes les histoires (sauf sur la figure 4.2i), le premier processus écrit 1 (opération w(1)) puis lit une infinité de fois (opération r), le deuxième processus lit une fois puis écrit 2 et le troisième processus écrit 3 puis lit une infinité de fois.

La pile. Pour la pile, chaque valeur retournée par l'opération *pop* dépend sémantiquement d'un seul appel de l'opération *push*, mais toutes les opérations effectuées entre ces deux événements (et l'ordre de ces opérations) sont importantes pour déterminer si cette valeur de retour de l'opération *pop* est valide. La dépendance sémantique passe donc par une chaîne d'événements.

Approche. *Plutôt que de chercher une définition générale des dépendances sémantiques, notre approche est inverse : nous demandons l'existence d'un ordre partiel tel que les événements de l'histoire vérifient des propriétés en rapport avec la spécification séquentielle de l'objet. Un effet de bord de ces propriétés est que l'ordre partiel doit nécessairement contenir les dépendances sémantiques sans que nous ayons besoin de les définir.*

De légères différences dans les propriétés demandées à l'ordre partiel définissent des critères différents. Nous étudions donc non seulement la cohérence causale, extension de la mémoire causale, mais également d'autres variantes de ce critère.

Contribution. *Ce chapitre définit quatre nouveaux critères de cohérence.*

- *La cohérence causale faible peut être vue comme le plus petit dénominateur commun aux trois critères suivants. La conjonction de la cohérence causale faible et de la convergence ou de la cohérence pipeline forme de nouveaux critères de cohérence.*
- *La convergence causale est obtenue en conjuguant la cohérence causale faible et la convergence. C'est un critère plus fort que la cohérence d'écritures forte.*
- *La cohérence causale renforce la cohérence pipeline. Si on l'applique à la mémoire, elle correspond sous certaines hypothèses à la mémoire causale.*
- *L'implémentation standard de la cohérence causale en passage de messages implémentent légèrement plus que la cohérence causale. Les comportements autorisés par le critère de cohérence mais qui ne sont pas admises par les implémentations sont souvent appelées « fausse causalité ». Nous introduisons la* cohérence causale forte *qui renforce la cohérence causale en prenant en compte la fausse causalité.*

La figure 4.2 illustre leurs différences sur de petits exemples sur des flux fenêtrés de taille 2 ($\mathcal{W}_2$). La figure 4.2a résume tous ces critères et leur force relative. Le contenu de ce chapitre a été publié à la conférence PPoPP 2016 [94].

La suite de ce chapitre est organisée comme ceci. La section 4.1 présente le concept d'*ordre causal*, central dans tous les critères liés à la cohérence causale, ainsi que la cohérence causale faible et la convergence causale. La section 4.2 présente la cohérence causale et la cohérence causale forte, ainsi que le lien entre la cohérence causale et la mémoire causale. Enfin, la section 4.3 présente des contextes applicatifs dans lesquels les critères causaux se comportent comme la cohérence séquentielle.

4.1 La causalité comme critère de cohérence

4.1.1 Ordre causal et cônes temporels

La causalité est souvent perçue comme une propriété du système : la diffusion causale (voir section 2.1.2 page 36) est une primitive de communication selon laquelle deux messages ordonnés selon la relation « s'est produit avant » de Lamport (souvent elle-même désignée par le nom de *causalité*) seront reçus dans le même ordre par tous les processus. Cela n'entre pas en contradiction avec le fait d'en faire un critère de cohérence, c'est-à-dire une propriété des objets implémentés sur le système : du point de

vue du programme qui utilise l'objet partagé causalement cohérent, l'objet fait partie du système.

La cohérence causale est une façon d'affaiblir la cohérence séquentielle. Le but de la cohérence séquentielle est de fournir un ordre total sur toutes les opérations d'une histoire concurrente. Cet ordre total fixe une référence temporelle partagée à laquelle se réfèrent les événements de tous les processus. La cohérence causale affaiblit la cohérence séquentielle en autorisant le temps à être une notion relative à chaque processus : la relation d'ordre qui lie tous les événements n'est qu'un ordre partiel, appelé *ordre causal*. Comme l'ordre total de la cohérence séquentielle, l'ordre causal n'est pas imposé par des conditions extérieures. Il est seulement demandé l'*existence* d'un ordre causal pour expliquer les histoires concurrentes. En particulier, l'ordre causal n'est pas nécessairement unique.

Nous définissons maintenant formellement ce qu'est un ordre causal (définition 4.1). Un ordre causal est un ordre partiel qui contient l'ordre séquentiel $\mapsto$ de chaque processus, et tel que le futur causal d'un événement est cofini (son complémentaire est fini). Ce dernier point signifie qu'aucun événement ne peut être indéfiniment ignoré par un processus. Cela correspond à l'hypothèse que les messages finissent toujours par arriver dans les systèmes répartis. Il y a trois raisons pour lesquelles cela est important dans notre modèle.

1. Sans cette restriction, l'ordre causal pourrait être l'ordre des processus, qui est lui-même un ordre partiel. Les critères obtenus seraient alors beaucoup plus faibles car rien ne forcerait les processus à interagir. De tels critères seraient alors plus faibles que la cohérence locale et pourraient donc être implémentés trivialement, chaque processus mettant à jour et lisant sa propre copie locale. Cependant, de tels objets n'auraient aucune utilité dans les systèmes répartis.

2. Il est courant de dire que la cohérence causale est plus forte que la cohérence pipeline. D'après la définition de cette dernière (page 62), toutes les écritures doivent apparaître dans les linéarisations de tous les processus. Cette propriété est équivalente à l'hypothèse faite sur l'ordre causal.

3. Nous cherchons également à définir la convergence causale d'après laquelle la convergence doit être atteinte quand tous les processus ont les mêmes événements dans leur passé causal. Pour qu'un tel critère soit plus fort que la convergence, il est nécessaire que tous les processus finissent par avoir toutes les écritures dans leur passé causal.

Les contraintes supplémentaires vérifiées par l'ordre causal définiront les différents critères de cohérence.

Définition 4.1. Soit $H = (\Sigma, E, \Lambda, \mapsto, \nearrow)$ une histoire concurrente. Un *ordre causal* est un ordre partiel $\dashrightarrow$ sur les événements de E_H, qui contient $\mapsto$ et tel que pour tout $e \in E_H$, $\{e' \in E_H : e \not\dashrightarrow e'\}$ est fini.

L'ensemble des ordres causaux sur H est noté $co(H)$.

Remarque 4.1. Par la suite, on considérera l'histoire $H^{\dashrightarrow}$, qui contient les mêmes événements que H, mais dans laquelle l'ordre des processus a été remplacé par l'ordre causal $\dashrightarrow$. Pour cela, il est important que l'ordre causal soit un bel ordre (voir page 34). Il est facile de montrer que toute relation d'ordre qui contient un bel ordre est également un bel ordre. Comme l'ordre de processus est un bel ordre, l'ordre causal est également un bel ordre.

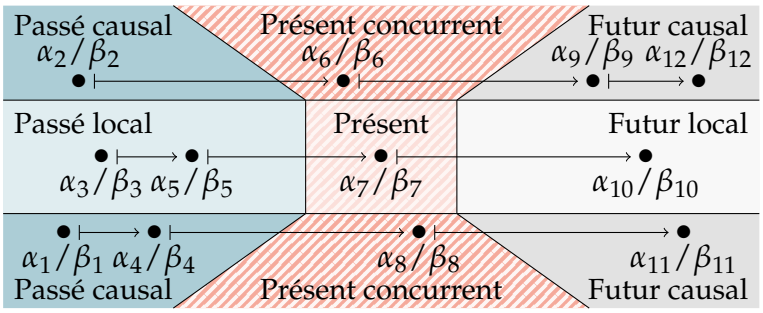


FIGURE 4.3 – Un ordre causal permet de diviser, pour chaque événement, l'histoire en six zones temporelles.

Une histoire concurrente possédant un ordre causal est décrite par les deux ordres partiels $\mapsto$ et $\dashrightarrow$. Pour chaque élément e, chaque ordre partiel sur l'ensemble des événements définit un cône temporel (similaire au cône de lumière dans la relativité restreinte) que l'on peut séparer en quatre zones : le passé qui contient les prédécesseurs de e, le présent réduit à e, le futur qui contient les successeurs de e et l'extérieur du cône qui contient les événements incomparables à e. Comme $\mapsto \subset \dashrightarrow$, le cône local au processus est emboîté dans celui décrit par la causalité. Du point de vue de e, l'histoire concurrente est donc partitionnée en six zones représentées sur la figure 4.3 :

- le *présent* $\{e\}$;
- le *passé local* $\lfloor e \rfloor_{\mapsto} \setminus \{e\}$, qui correspond au passé selon l'ordre de processus strict ;
- le *futur local* $\lceil e \rceil_{\mapsto} \setminus \{e\}$, le symétrique du passé local dans le futur ;
- le *passé causal* strict $\lfloor e \rfloor_{\dashrightarrow} \setminus \lfloor e \rfloor_{\mapsto}$, la partie du passé causal qui n'est pas déjà contenue dans le passé local ni dans le présent ;
- le *futur causal* strict $\lceil e \rceil_{\dashrightarrow} \setminus \lceil e \rceil_{\mapsto}$, l'équivalent du passé causal strict dans le futur ;
- le *présent concurrent* $E \setminus (\lfloor e \rfloor_{\dashrightarrow} \cup \lceil e \rceil_{\dashrightarrow})$, à l'extérieur des deux cônes.

La façon dont le présent est affecté par ces différentes zones décrit différents critères de cohérence.

4.1.2 La cohérence causale faible

L'existence seule d'un ordre causal sur une histoire concurrente H qui vérifie la définition 4.1 n'apporte absolument aucune information sur cette histoire pour deux raisons.

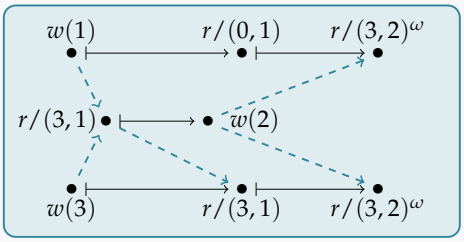
1. Prenons l'analogie de la cohérence séquentielle : il ne suffit pas d'ordonner totalement les événements d'une histoire concurrente pour la rendre séquentiellement cohérente ; la linéarisation correspondant à l'ordre total proposé doit également satisfaire la spécification séquentielle de l'objet. Il en va de même pour l'ordre causal : de la même façon que toute relation d'ordre peut être étendue en ordre total, un ordre causal peut être trouvé pour toute histoire concurrente (par exemple, toute extension linéaire en est un, d'après la remarque 4.1). La première question qui se pose est donc la façon de relier l'ordre causal et les valeurs que les opérations peuvent retourner pour donner un sens à cet ordre causal.

2. La relation de dépendance entre les lectures et les écritures est une notion centrale dans la définition originale de la mémoire causale. La deuxième question est la façon de forcer l'ordre causal à contenir ces dépendances.

COHÉRENCE CAUSALE FAIBLE

✗ Non composable p. 41
✓ Décomposable p. 41
✓ Faible p. 126

La cohérence causale faible assure que chaque lecture retourne une valeur en accord avec une connaissance complète de son passé causal, et seulement de celui-ci. Ainsi, dans l'histoire de la figure 4.2h (rappelée ci-contre), l'écriture $r(3)$ du troisième processus est nécessairement dans le passé causal du deuxième processus lorsqu'il lit $(3,1)$ — sinon d'où viendrait le 3 qu'il retourne ? De plus, comme l'ordre causal doit contenir l'ordre de processus, la lecture du deuxième processus précède son écriture, et par transitivité, l'écriture du 3 précède l'écriture du 2. C'est la raison pour laquelle les deux processus convergent vers $(3,2)$ et non $(2,3)$.



Formellement, une histoire H est faiblement causalement cohérente par rapport à un type de données abstrait T (définition 4.2) s'il existe un ordre causal $\dashrightarrow$ tel que, pour chaque événement e, il existe une linéarisation du passé causal de e qui mène dans un état dans lequel e est possible, c'est-à-dire une linéarisation de l'histoire $H^{\dashrightarrow}[\lfloor e \rfloor_{\dashrightarrow}/\{e\}]$ qui contient e et les écritures de son passé causal ordonnées selon l'ordre causal. Dans l'exemple ci-dessus, l'ordre causal est représenté par toutes les flèches (pleines et tirets) et les linéarisations suivantes correspondent à ces critères pour les premières lectures des trois processus :

$$w(1)\cdot r/(0,1) \qquad w(3)\cdot w(1)\cdot r/(3,1) \qquad w(3)\cdot w(1)\cdot r\cdot r/(3,1)$$

Définition 4.2 (Cohérence causale faible)**.** La *cohérence causale faible* est le critère de cohérence

$$WCC : \left\{ \begin{array}{ccc} \mathcal{T} & \to & \mathcal{P}(\mathcal{H}) \\ T & \mapsto & \left\{ H \in \mathcal{H} : \begin{array}{l} \exists \dashrightarrow \in co(H), \\ \forall e \in E_H, \mathrm{lin}\left(H^{\dashrightarrow}[\lfloor e \rfloor_{\dashrightarrow}/\{e\}]\right) \cap L(T) \neq \emptyset \end{array} \right\} \end{array} \right.$$

L'histoire de la figure 4.2d (à droite) n'est pas faiblement causalement cohérente. En effet, la lecture $(0,1)$ du deuxième processus a l'écriture du 1 dans son passé causal et l'écriture du 2 dans son futur causal. On a donc $w(1) \dashrightarrow w(2)$. Or il faut que l'écriture du 2 précède celle du 1 dans la linéarisation de la lecture $(2,1)$, ce qui est en contradiction avec l'ordre causal. Cette lecture est donc impossible.

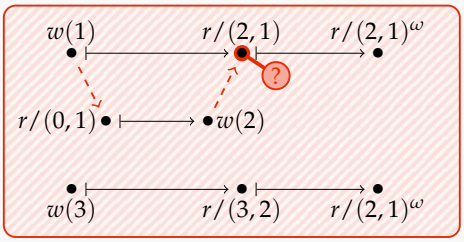


FIGURE 4.4 – La cohérence causale faible.

La cohérence causale faible (*WCC*, pour « weak causal consistency »), définie formellement dans la figure 4.4, répond à ces deux problèmes de la façon suivante : toute lecture doit être compatible avec un état qui résulte de l'exécution séquentielle de toutes les écritures de son passé causal, et seulement celles-ci, dans un ordre qui respecte l'ordre causal.

Cette définition force bien l'ordre causal à contenir les relations de dépendance entre les lectures et les écritures dans le cas de la mémoire faiblement causalement cohérente : si un processus lit une valeur v dans un registre x, l'exécution séquentielle des écritures de son passé causal doit mener à un état dans lequel x vaut v, mais cela n'est possible que si le passé causal contient une écriture de v dans x. Nous étudierons plus en détails le cas de la mémoire causale dans la section 4.2.2.

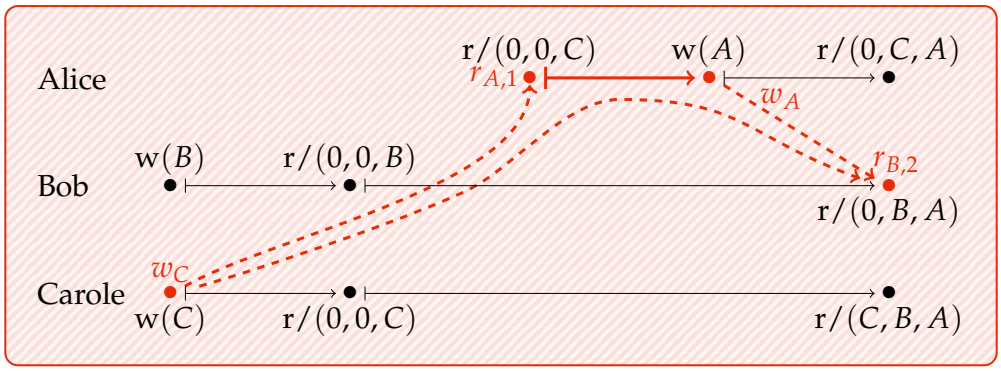


FIGURE 4.5 – L'histoire du préambule n'est pas faiblement causalement cohérente.

Voyons maintenant comment la cohérence causale faible s'applique à l'exemple d'Alice, Bob et Carole présenté au début de ce chapitre. La figure 4.5 modélise l'histoire grâce à un flux fenêtré de taille trois. L'envoi d'un message est modélisé par l'écriture d'une valeur A, B ou C, et la lecture de la file de messages est une lecture du flux fenêtré. Essayons de construire un ordre causal $\dashrightarrow$ qui vérifie la propriété demandée. Lors de sa première lecture (événement $r_{A,1}$), Alice voit le message de Carole. Toute linéarisation conforme à la spécification séquentielle de la file de messages qui termine par cette lecture doit nécessairement contenir l'envoi du message par Carole (événement w_C). Comme les écritures de cette linéarisation sont exactement celles du passé causal de la lecture, on a $w_C \dashrightarrow r_{A,1}$. Par le même raisonnement sur l'envoi du message par Alice (événement w_A) et la première lecture où Bob voit le message d'Alice (événement $r_{B,2}$), on peut déduire que $w_A \dashrightarrow r_{B,2}$. De plus, on sait d'après la définition 4.1 que l'ordre causal contient l'ordre de programme, et donc que la première lecture d'Alice précède causalement l'envoi de son message. Mais alors, $w_C \dashrightarrow r_{A,1} \dashrightarrow w_A \dashrightarrow r_{B,2}$ donc la linéarisation demandée pour la lecture $r_{B,2}$ de Bob doit nécessairement contenir l'événement w_C, ce qui ne correspond pas à la lecture observée. La cohérence causale faible permet donc bien de spécifier le comportement voulu.

4.1.3 La convergence causale

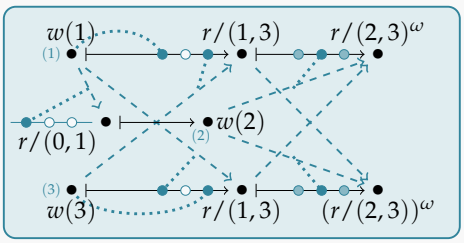
Il y a plusieurs façons de renforcer la convergence par la cohérence causale faible. La convergence causale (*CCv* pour « causal convergence »), définie formellement sur la figure 4.6, renforce la cohérence d'écritures forte en remplaçant la relation de visibilité par un ordre causal. Les écritures sont donc totalement ordonnées par un ordre $\leq$ qui est respecté par les linéarisations de la cohérence causale faible pour toutes les lectures. Ainsi, si les processus font un nombre fini d'écritures, le nombre de lectures qui n'auront pas toutes les écritures dans leur passé causal sera également fini. Toutes les autres lectures auront toutes les écritures dans leur passé causal, ordonnées dans le même ordre. Elles seront donc exécutées dans le même état.

L'algorithme de la figure 4.14 fonctionne de la même manière que l'algorithme de la figure 3.7 pour la cohérence d'écritures forte. Le processus p_i gère une horloge de Lamport $vtime_i$ qui lui permet d'estampiller toutes les opérations pour les ordonner totalement, et une variable $history_i$ qui contient toute l'histoire. Pour lire l'état courant, p_i exécute toute l'histoire en partant de l'état initial. Pour exécuter une opération d'écriture α, il diffuse causalement α et l'estampille permettant de l'ordonner. À la réception d'un tel message, p_i insère simplement l'événement dans sa variable $history_i$. La seule différence avec l'algorithme de la figure 3.7 est l'utilisation de la diffusion causale, qui permet d'assurer la transitivité de la relation de visibilité.

CONVERGENCE CAUSALE

✗ Non composable p. 41
✓ Décomposable p. 41
✓ Faible p. 126

La convergence causale renforce à la fois la cohérence d'écritures forte et la cohérence causale faible. Pour cela, elle renforce la cohérence d'écritures forte en remplaçant la relation de visibilité par un ordre causal (c'est-à-dire en imposant la transitivité à la relation de visibilité). L'histoire de la figure 4.2g (rappelée ci-contre) est causalement convergente : l'ordre causal est figuré en tirets et l'ordre total de la cohérence d'écritures forte est représenté par les cercles bleus se remplissant quand les écritures deviennent visibles.



Formellement, une histoire H est causalement convergente par rapport à un type de données abstrait T (définition 4.3) s'il existe un ordre causal $\dashrightarrow$ et un ordre total $\leq$ qui contient l'ordre causal tel que chaque lecture résulte de l'exécution des écritures de son passé causal, triées selon l'ordre total. Dans l'histoire ci-dessus, les écritures sont ordonnées $w(1) \leq w(2) \leq w(3)$ et l'un des ordres causaux possibles peut être inféré d'après les flèches en tirets. Les lectures correspondent aux linéarisations :

$$w(1)\cdot r/(0,1) \qquad w(1)\cdot w(3)\cdot r/(1,3) \qquad w(1)\cdot w(2)\cdot w(3)\cdot r\cdot r/(2,3)$$

Définition 4.3 (convergence causale)**.** La *convergence causale* est le critère de cohérence

$$CCv : \left\{ \begin{array}{ccc} \mathcal{T} & \to & \mathcal{P}(\mathcal{H}) \\ T & \mapsto & \left\{ H \in \mathcal{H} : \begin{array}{c} \exists \dashrightarrow \in co(H), \exists \leq \text{ ordre total sur } E_H, \dashrightarrow \subset \leq \\ \wedge \quad \forall e \in E_H, \text{lin}\left(H^{\leq}[\lfloor e \rfloor_{\dashrightarrow}/\{e\}]\right) \cap L(T) \neq \emptyset \end{array} \right\} \end{array} \right.$$

L'histoire de la figure 4.2h (ci-contre) n'est pas causalement convergente. En effet, pour respecter la convergence causale, l'écriture $w(1)$ doit précéder causalement la lecture $r/(3,1)$ du deuxième processus, qui elle-même précède causalement l'écriture $w(2)$. Selon l'ordre total, on doit donc avoir $w(1) \leq w(2)$. De plus, les lectures de $(1,3)$ imposent que $w(3) \leq w(1)$. Après convergence, les lectures devraient retourner $(2,3)$ et non $(3,2)$.

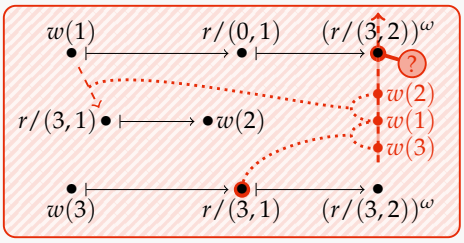


FIGURE 4.6 – La convergence causale.

Proposition 4.1. *Toute histoire concurrente admise par l'algorithme de la figure 4.7 est causalement convergente.*

Démonstration. Soient $T \in \mathcal{T}$ et H une histoire admise par l'algorithme de la figure 4.7. Soient deux événements $e, e' \in E_H$ invoqués par les processus p_i et $p_{i'}$ dans les états respectifs $(vtime_i, history_i)$ et $(vtime_{i'}, history_{i'})$. Nous posons :

- $e \dashrightarrow e'$ si $e = e'$ ou si le processus exécutant e' a reçu le message envoyé pendant l'exécution de e', avant d'exécuter e. Comme la réception est causale, $\dashrightarrow$ est bien un ordre partiel. Comme tout message est reçu instantanément par son émetteur, $\mapsto \subset \dashrightarrow$. Comme tout message finit par être acheminé, $\dashrightarrow$ est bien un ordre causal.
- $e \leq e'$ si $vtime_i < vtime_{i'}$ ou $vtime_i = vtime_{i'}$ et $i < i'$. Cet ordre lexicographique est total car deux opérations du même processus ont des horloges différentes et l'identifiant d'un processus est unique. De plus, si $e \dashrightarrow e'$, d'après les lignes 9 et 13, $vtime_{i'} \leq vtime_i + 1$ donc $e \leq e'$.

```
1  algorithm CCv(A, B, Z, ζ0, τ, δ)
2  |  variable vtime_i ∈ ℕ ← 0 ;                          // horloge linéaire
3  |  variable history_i ⊂ (ℕ × ℕ × A) ← ∅ ;          // liste des événements
4  |  operation apply (α ∈ A) ∈ B
5  |  |  variable state ∈ Z ← ζ0 ;
6  |  |  for (t_j, j, α') ∈ history_i triés selon (t_j, j) do          // lecture
7  |  |  |  state ← τ(state, α') ;                  // l'histoire est rejouée
8  |  |  end
9  |  |  causal broadcast mUpdate (vtime_i + 1, i, α) ;          // écriture
10 |  |  return δ(state, α) ;
11 |  end
12 |  on receive mUpdate (t_j ∈ ℕ, j ∈ ℕ, α ∈ A)
13 |  |  vtime_i ← max(vtime_i, t_j) ;
14 |  |  history_i ← history_i ∪ {(t_j, j, α)} ;
15 |  end
16 end
```

FIGURE 4.7 – Algorithme générique $CCv(T)$: code pour p_i

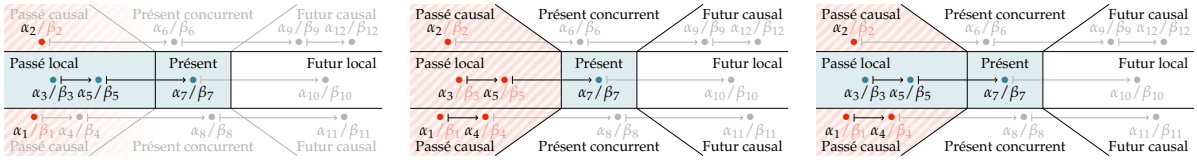
Soit $e \in E_H$. Les lignes 6 et 7 construisent explicitement une linéarisation de $H^{\leq}[\lfloor e \rfloor_{\dashrightarrow}/\{e\}]$ par définition de $\leq$ et $\dashrightarrow$ qui fait partie de $L(T)$ par définition de T. □

4.2 La cohérence causale

La mémoire causale [6] est plus forte que la mémoire PRAM [76]. Pour que la cohérence causale soit une extension de la mémoire causale aux autres types de données abstraits, il faut donc que la cohérence causale soit plus forte que la cohérence pipeline. La cohérence causale faible et la cohérence pipeline sont incomparables (les figures 4.2d et 4.2g montrent des contre-exemples dans les deux sens). Dans cette section, nous présentons la cohérence causale, un nouveau critère de cohérence plus fort à la fois que la cohérence pipeline et que la cohérence causale faible.

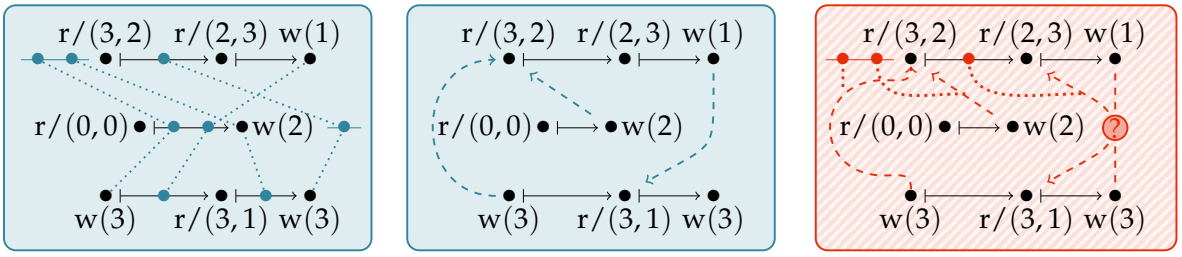
4.2.1 Définition

La différence entre cohérence causale faible et cohérence pipeline peut être expliquée en termes de zones temporelles, comme sur la figure 4.8. D'un côté, la cohérence pipeline cherche à imposer une vue cohérente à chaque processus pendant toute sa durée de vie. Par conséquent, le présent doit prendre en compte tout son passé local, lectures comme écritures. Par contre, elle ne garantit pas l'existence d'un ordre causal, donc la frontière entre les écritures des autres processus prises en compte et celles ignorées est floue (figure 4.8a). D'un autre côté, la cohérence causale faible cherche à forcer le respect de l'ordre causal et requiert donc que le présent soit conforme à toutes les écritures du passé causal (figure 4.8b). La cohérence causale (*CC* pour « causal consistency ») renforce la cohérence causale faible et la cohérence pipeline en considérant les événements du passé local comme dans la cohérence pipeline et ceux du passé causal (qui ne font pas partie du passé local) comme dans la cohérence causale. Plus précisément, pour chaque événement, il doit exister une linéarisation contenant les écritures de son passé



(a) Cohérence pipeline. (b) Cohérence causale faible. (c) Cohérence causale.

FIGURE 4.8 – Les différences entre la cohérence pipeline, la cohérence causale faible et la cohérence causale peuvent être expliquées en termes de zones temporelles. Plus les événements du passé imposent de contraintes sur le présent plus le critère obtenu est fort. En rouge hachuré sont représentés les événements dont seule la partie écriture a une influence sur le présent, et en bleu les événements dont la valeur retournée ne doit pas entrer en conflit avec le présent.



(a) Cohérence pipeline. (b) Cohérence causale faible. (c) Cohérence causale.

FIGURE 4.9 – La cohérence causale est strictement plus forte que la conjonction de la cohérence pipeline et de la cohérence causale faible.

causal ainsi que les lectures de son passé local (figure 4.8c). La définition formelle est donnée sur la figure 4.10.

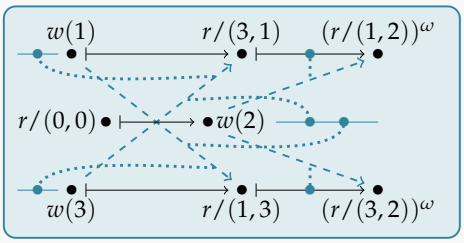
La cohérence causale est plus forte que la cohérence causale faible car elle ne fait que rajouter des contraintes. Il n'est en revanche pas trivial que la cohérence causale est plus forte que la cohérence pipeline (*PC*) étant données les définitions : l'existence d'une linéarisation pour chaque événement ne donne pas directement une linéarisation pour toute l'histoire. Nous prouvons la proposition 4.2, légèrement plus forte, qui sera utile dans la section 4.2.2. Le fait que $PC \leq CC$ en est un corollaire direct car $\mapsto \subset \dashrightarrow$.

La cohérence causale est plus que l'addition exacte de la cohérence causale faible et de la cohérence pipeline : l'histoire de la figure 4.2i, reprise sur la figure 4.9 est un contre-exemple. Cette histoire vérifie bien la cohérence pipeline : la figure 4.9a montre comment obtenir les linéarisations pour les deux processus. Elle est également faiblement causalement cohérente : la figure 4.9b montre un ordre causal possible. Cependant, elle n'est pas causalement cohérente car la deuxième lecture du premier processus pose problème. La linéarisation pour cette lecture contient forcément au moins les événements étiquetés $w(3)$, $w(2)$, $r/(3,2)$, $w(3)$ et $r/(2,3)$ dans cet ordre, donc la dernière écriture du troisième processus fait nécessairement partie du passé causal de cette lecture. Cela est pourtant impossible car cela créerait un cycle dans l'ordre causal. Le problème ne se posait pas avec la cohérence causale faible car celle-ci autorise les mêmes écritures de 2 et de 3 à être vues dans un ordre différent entre la première et la deuxième lecture du premier processus.

Cohérence Causale

✗ Non composable p. 41
✓ Décomposable p. 41
✓ Faible p. 126

La cohérence causale renforce à la fois la cohérence causale faible et la cohérence pipeline. La cohérence causale faible impose que chaque lecture retourne une valeur en rapport avec son passé causal et la cohérence pipeline empêche que les lectures d'un même processus soient indépendantes les unes des autres : dans la cohérence causale, une lecture doit donc prendre en compte à la fois les écritures de son passé causal et les lectures de son passé local. Par exemple, dans l'histoire de la figure 4.2e (rappelée ci-contre), les écritures du 1 et du 2 sont causalement indépendantes, donc la dernière lecture du premier processus pourrait aussi bien être $(2,1)$ que $(1,2)$ selon la cohérence causale faible. Or la spécification séquentielle interdit de passer de l'état $(3,1)$ à l'état $(2,1)$, d'où la lecture $(1,2)$ dans cette histoire.



Formellement, une histoire H est causalement cohérente par rapport à un type de données abstrait T (définition 4.4) s'il existe un ordre causal $\dashrightarrow$ tel que, pour chaque événement e, il existe une linéarisation des écritures du passé causal et des lectures du passé local de e qui mène dans un état dans lequel e est possible, c'est-à-dire une linéarisation de l'histoire $H^{\dashrightarrow}[\lfloor e \rfloor_{\dashrightarrow} / \lfloor e \rfloor_{\mapsto}]$ qui contient les écritures du passé causal $\lfloor e \rfloor_{\dashrightarrow}$ et les lectures du passé local $\lfloor e \rfloor_{\mapsto}$ de e, ordonnées selon l'ordre causal. Dans l'exemple ci-dessus, les linéarisations suivantes correspondent à ces critères pour les deux premières lectures du premier processus, où les valeurs muettes $\bot$ sont laissées visibles pour différencier les écritures cachées de celles qui ne le sont pas :

$$w(3)/\bot \cdot w(1) \cdot r/(3,1) \qquad\qquad w(3)/\bot \cdot w(1) \cdot r/(3,1) \cdot r \cdot w(2)/\bot \cdot r/(1,2)$$

Définition 4.4 (Cohérence causale). La *cohérence causale* est le critère de cohérence

$$CC : \left\{ \begin{array}{ccc} \mathcal{T} & \to & \mathcal{P}(\mathcal{H}) \\ T & \mapsto & \left\{ H \in \mathcal{H} : \begin{array}{l} \exists \dashrightarrow \in co(H), \\ \forall e \in E_H, \mathrm{lin}\left(H^{\dashrightarrow}[\lfloor e \rfloor_{\dashrightarrow} / \lfloor e \rfloor_{\mapsto}]\right) \cap L(T) \neq \emptyset \end{array} \right\} \end{array} \right.$$

L'histoire de la figure 4.2i (à droite) n'est pas causalement cohérente, bien qu'elle vérifie à la fois la cohérence causale faible et la cohérence pipeline. Cela est dû au fait que les deux écritures de la valeur 3 sont interprétées de manière différente par les deux critères. Dans la cohérence pipeline, la linéarisation pour le premier processus place la deuxième écriture du 3 entre les lectures $(3,2)$ et $(2,3)$, ce qui crée un cycle dans l'ordre causal entre cette écriture et l'écriture du 1. Dans la cohérence causale faible, les lectures $(3,2)$ et $(2,3)$ ont les mêmes écritures dans leur passé causal, mais le changement spontané d'état viole la spécification séquentielle. Ceci montre que la cohérence causale est plus que la simple conjonction de la cohérence causale faible et de la cohérence pipeline.

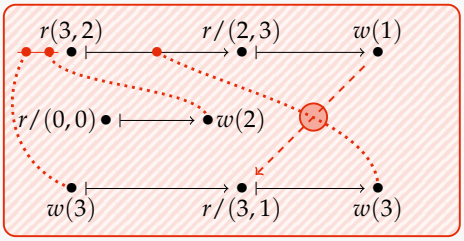


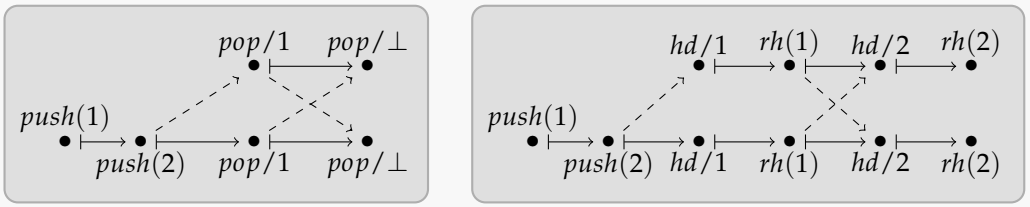
FIGURE 4.10 – La cohérence causale.

Proposition 4.2. *Soient $T \in \mathcal{T}$ un ADT et $H \in CC(T)$ une histoire causalement cohérente. On a $\forall p \in \mathcal{P}_H, \mathrm{lin}\left(H^{\dashrightarrow}[E_H / p]\right) \cap L(T) \neq \emptyset$.*

Démonstration. Soient $T \in \mathcal{T}$, $H \in CC(T)$ et $p \in \mathcal{P}_H$.

Si p est fini, il possède un plus grand élément e. Comme H est causalement cohérente, il existe une linéarisation $l_e \in \mathrm{lin}(H^{\dashrightarrow}[\lfloor e \rfloor_{\dashrightarrow} / \lfloor e \rfloor_{\mapsto}]) \cap L(T)$. Comme e est maximal, $\lfloor e \rfloor_{\mapsto} = p$. Comme $\mapsto \subset \dashrightarrow$, il existe une linéarisation l de $H^{\dashrightarrow}[E_H / p]$ qui a l_e pour préfixe. Comme $l_e \in L(T)$ et tous les événements qui sont dans l et pas dans l_e sont cachés, $l \in L(T)$.

QU'EST-CE QU'UNE FILE CAUSALE ?



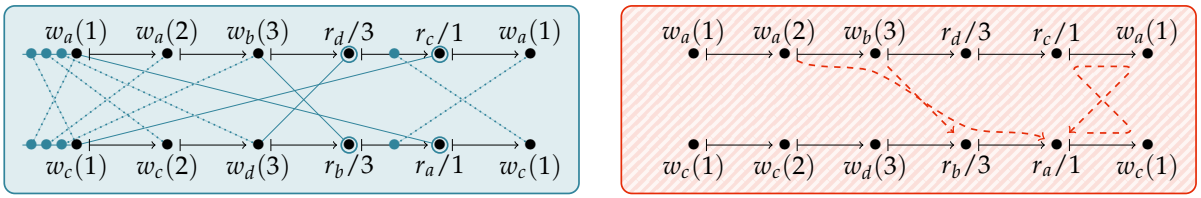
(a) L'ADT file $\mathcal{Q}_{\mathbb{N}}$ (b) L'UQ-ADT file $\mathcal{Q}'_{\mathbb{N}}$

Les histoires des figures 4.11a et 4.11b sont toutes les deux causalement cohérentes, mais pas séquentiellement cohérentes. Dans l'histoire de la figure 4.11a, quand les deux processus appellent l'opération pop, ils sont dans le même état $[1, 2]$ *et donc obtiennent* 1*. Ensuite, ils intègrent le fait que l'autre processus a supprimé la tête de la file, qu'ils pensent être le* 2 *restant. Lors du pop suivant, la file est vide. Aucun critère faible ne peut assurer que chaque élément inséré sera tiré une et une seule fois même si une infinité d'opérations pop sont effectuées. Cependant, cet exemple montre que la cohérence causale ne peut garantir ni l'existence (*2 *n'est jamais lu) ni l'unicité (*1 *est lu deux fois) d'une lecture pour chaque élément. La raison est que la cohérence causale — comme tous les critères faibles présentés dans cette thèse — affaiblit la cohérence séquentielle en donnant un rôle différent aux lectures et aux écritures, ce qui exclut l'atomicité des opérations qui sont à la fois des lectures et des écritures comme pop. Sur la figure 4.11b, l'opération pop a été séparée en une opération de lecture hd qui lit la tête sans y toucher et une opération d'écriture rh qui la supprime (type* $\mathcal{Q}'_{\mathbb{N}}$ *définition 2.9). L'histoire est similaire au cas précédent et les deux processus lisent* 1 *puis effectuent* $rh(1)$*. Ce faisant, ils ne suppriment pas* 2 *en tête de file. Cette technique permet de garantir que toutes les insertions correspondent à au moins une lecture.*

FIGURE 4.11 – Les critères de cohérence faible ne garantissent pas l'atomicité des parties lecture et écriture des opérations. Utiliser un UQ-ADT peut être une bonne solution pour garder le contrôle.

Sinon, p est infini et il ne possède pas de plus grand élément. À la place, on construit une suite croissante de linéarisations $(l_n)_{n\in\mathbb{N}}$ qui converge vers une histoire séquentielle ayant la propriété recherchée. Les linéarisations données par la cohérence causale pour les événements successifs ne sont pas nécessairement préfixes les unes des autres (voir la figure 4.13). Les linéarisations contiennent donc, en plus du passé des événement, une partie de leur présent concurrent. Les événements de p sont numérotés dans l'ordre $e_1 \mapsto e_2 \mapsto \ldots$ et nous définissons pour tout $n \geq 1$ l'ensemble L_n tel que $l.e_n \in L_n$ si et seulement si il existe l' tel que $l.e_n.l' \in \mathrm{lin}(H^{\dashrightarrow}[\{e_n\} \cup E_H \setminus \lceil e_n \rceil_{\dashrightarrow} / \lfloor e_n \rfloor_{\mapsto}]) \cap L(T)$. En d'autres termes, L_n contient les linéarisations du passé causal et du présent concurrent de e_n, tronquées après e_n. L_n n'est pas vide car il contient toutes les linéarisations données par la cohérence causale. Il est aussi fini car $\dashrightarrow$ est un ordre causal, donc $E_H \setminus \lceil e \rceil_{\dashrightarrow}$ est fini. Notons également que toutes les linéarisations de L_{n+1} ont un préfixe dans L_n car $e_n \dashrightarrow e_{n+1}$. $L(T)$ est donc clos par prise du préfixe.

Nous construisons maintenant par induction une suite $(l_n)_n$ de mots de L_n tels que pour tout n, l_n est un préfixe de l_{n+1}. La chaîne vide $l_0 = \varepsilon$ est préfixe d'un mot de L_n pour tout n. Supposons que pour un n donné, on a réussi à trouver un mot $l_n \in L_n$ qui est préfixe d'un mot de L_k pour tout $k > n$. Montrons par l'absurde qu'il existe un tel mot dans L_{n+1}, qui a l_n comme préfixe. Si ce n'était pas le cas, comme L_{n+1} est fini, il existerait un k pour lequel aucun mot $l \in L_n$ ne serait un préfixe d'un mot de L_k. Celui-ci n'étant pas vide, cela est en contradiction avec le fait que tous les mots de L_k ont un préfixe dans L_n. Nous avons donc construit une suite $(l_n)_n$ de mots de L_n tels



(a) Mémoire causale. (b) Cohérence causale.

FIGURE 4.12 – La cohérence causale est différente de la mémoire causale.

que pour tout n, l_{n+1} est un préfixe de l_n. La suite $(l_n)_n$ converge vers un mot infini l. Comme tous les préfixes de l sont dans $L(T)$, $l \in L(T)$. De plus, comme $\dashrightarrow$ est un ordre causal, tous les événements sont dans le passé causal de e_n pour un certain n, donc l contient tous les événements de $H[E_H/p]$. Comme l'ordre causal est respecté par tous les préfixes de l qui contiennent deux événements e et e', il est également respecté dans l. Finalement, $l \in \text{lin}\,(H^{\dashrightarrow}[E_H/p]) \cap L(T)$.

□

4.2.2 Étude de cas : la mémoire causale

Il est important que la notion de cohérence causale que nous venons de définir soit similaire à celle de mémoire causale déjà existante, et dont on reprend le nom. La définition de la mémoire causale est donnée page 66. Dans cette section, nous comparons la cohérence causale appliquée à la mémoire et la mémoire causale. Nous montrons premièrement que ces deux notions sont différentes en exhibant une histoire acceptée par la mémoire causale mais pas causalement cohérente, dans laquelle la même valeur est écrite deux fois dans le même registre. Ensuite, nous montrons que la mémoire causale et la mémoire causalement cohérente sont identiques dans le cas où toutes les valeurs écrites sont différentes. Dans toute cette section, on considère un ensemble dénombrable de noms de registres X, et la mémoire $\mathcal{M}_X$ composée de ces registres.

La mémoire causale définit l'ordre causal explicitement en considérant une relation d'écriture-lecture dans un premier temps, puis en utilisant cet ordre causal de la même façon que l'ordre de programme dans la cohérence pipeline dans un deuxième temps. Cette approche montre deux limites. Premièrement, il n'y a pas unicité de la relation d'écriture-lecture. Deuxièmement, la construction des linéarisations des processus n'est que peu contrainte par l'ordre causal. Ces deux limites font qu'une lecture n'est pas obligée de retourner une valeur écrite par son antécédent direct dans la relation d'écriture-lecture. L'histoire de la figure 4.12 illustre ce point. Dans cette histoire, deux processus écrivent et lisent sur une mémoire composée de quatre registres. Le premier processus écrit 1 puis 2 dans le registre a, 3 dans b puis lit 3 dans d et 1 dans c et enfin écrit à nouveau 1 dans a. Le deuxième processus a un comportement similaire en échangeant les rôles des registres a et c et des registres b et d. Cette histoire vérifie la propriété de la mémoire causale : une relation d'écriture-lecture possible, dessinée en traits pleins terminés par des cercles sur la figure 4.12a, associe chaque lecture à la première écriture de la même valeur sur le même registre. Les linéarisations $w_a(1)/\bot.w_a(2)/\bot.w_b(3)/\bot.w_c(1).w_c(2).w_d(3).r_d/3.r_b.\ r_a.w_c(1).r_c/1.w_a(1)/\bot$ pour le premier processus et $w_a(1).w_a(2).w_b(3).w_c(1)/\bot.w_c(2)/\bot.w_d(3)/\bot.r_b/3.r_d.\ r_c.w_a(1).r_a/1.w_c(1)/\bot$ pour le deuxième processus, représentées en pointillés, sont correctes, donc l'histoire vérifie la propriété de la mémoire causale.

Pourtant, dans ces linéarisations, les valeurs lues par les deux dernières lectures ne sont pas le fruit de leur antécédent dans la relation d'écriture-lecture. Si l'on change cette relation pour rétablir les véritables dépendances de données, nous obtenons un cycle dans l'ordre causal. L'histoire n'est en fait pas causalement cohérente comme le montre la figure 4.12b. L'écriture $w_b(3)$ précède nécessairement la lecture $r_b/3$ dans l'ordre causal, donc le passé causal de la lecture $r_a/1$ contient les écritures $w_a(1)$ et $w_a(2)$ dans cet ordre. Par respect pour la spécification séquentielle, la deuxième écriture $w_a(1)$ doit être placée entre l'écriture $w_a(2)$ et la lecture $r_a/1$ dans la linéarisation de cette dernière, et donc dans l'ordre causal. Symétriquement, la dernière écriture sur le registre c doit précéder causalement la lecture $w_c(1)$, ce qui crée un cycle dans l'ordre causal.

Ce problème est une limite bien connue de la mémoire causale [81], due au fait que l'approche de la mémoire causale basée sur la sémantique des opérations n'est pas bien adaptée pour définir les critères de cohérence. La manière habituelle de résoudre ce problème est de faire l'hypothèse que toutes les valeurs écrites sont distinctes. Cela ne restreint pas l'utilisation de la mémoire car il est toujours possible d'ajouter de l'information (par exemple une estampille similaire à celles de la cohérence d'écritures forte) à la valeur écrite pour la rendre unique. Cependant, cette solution n'est pas idéale dans la spécification car elle change de façon artificielle les opérations fournies par l'objet, et donc la spécification séquentielle. Nous montrons maintenant que, sous l'hypothèse d'unicité des valeurs écrites, la mémoire causalement cohérente et la mémoire causale sont identiques. Cela signifie que notre définition de la cohérence causale règle le problème soulevé ci-dessus en restant égale à la mémoire causale dans tous les cas qui ne posent pas de problème.

Proposition 4.3. *Soit H une histoire concurrente. Si H est causalement cohérente par rapport à la mémoire sur X ($H \in CC(\mathcal{M}_X)$), alors H est $\mathcal{M}_X$-causale.*

Démonstration. Supposons que H est causalement cohérente. Pour tout événement e, il existe une linéarisation $l_e \in \text{lin}(H^{\dashrightarrow}[\lfloor e \rfloor_{\dashrightarrow} / \lfloor e \rfloor_{\mapsto}]) \cap L(\mathcal{M}_X)$. Remarquons que cette linéarisation n'est pas forcément unique, mais nous la fixons pour chaque événement dès maintenant. Nous définissons la relation d'écriture-lecture $\multimap$ par $e \multimap e'$ si $\Lambda(e') = \mathrm{r}_x/n$ et e est l'événement correspondant à la dernière écriture sur x dans l_e. Comme $l_e \in L(\mathcal{M}_X)$, $\Lambda(e) = \mathrm{w}_x(n)$. e' a au plus un antécédent par $\multimap$, et s'il n'en a pas $n = 0$.

La clôture transitive $\xrightarrow{CM}$ de $\multimap \cup \mapsto$ est un ordre partiel contenu dans $\dashrightarrow$. Par la proposition 4.2, pour tout $p \in \mathscr{P}_H$, $\text{lin}\,(H^{\dashrightarrow}[E_H/p]) \cap L(\mathcal{M}_X) \neq \emptyset$, donc H est $\mathcal{M}_X$-causale. □

Proposition 4.4. *Soit H une histoire concurrente telle que, pour tous $e, e' \in E_H$ avec $\Lambda(e) = w_x(n)/\perp$ et $\Lambda(e') = w_y(p)/\perp$, $e \neq e' \Rightarrow (x,n) \neq (y,p)$. Si H est $\mathcal{M}_X$-causale, alors $H \in CC(\mathcal{M}_X)$.*

Démonstration. Supposons que H est $\mathcal{M}_X$-causale. $\xrightarrow{CM}$ est un ordre causal. Soient $p \in \mathscr{P}_H$ et $e \in p$. Il existe une linéarisation $l_p \in \text{lin}(H^{\multimap CM}[E_H/p]) \cap L(\mathcal{M}_X)$, associée à un ordre total sur les événements $\leq_p$. Soit l_e l'unique linéarisation de $\text{lin}(H^{\leq_p}[\lfloor e \rfloor_{\dashrightarrow} / \lfloor e \rfloor_{\mapsto}]$. Soit $e' \in \lfloor e \rfloor_{\dashrightarrow}$ étiqueté r_x/n. Si e' n'a pas d'antécédent dans la relation d'écriture-lecture, $n = 0$. Sinon, cet antécédent e'' est la dernière écriture sur x avant e' dans l_p puisque c'est l'unique événement portant l'étiquette $w_x(n)$ dans H. Comme $e'' \xrightarrow{CM} e'$, c'est aussi la dernière écriture sur x avant e' dans l_e. Finalement, $l_e \in L(\mathcal{M}_X)$ donc H est causalement cohérente. □

4.2.3 Implémentation

Le problème de l'implémentation de la mémoire causale a été largement étudié dans la littérature [16,17,55,80]. L'algorithme générique de la figure 4.14 étend ces travaux à tous les types de données abstraits. Chaque processus p_i gère une variable locale $state_i$ qui reflète l'état courant de l'objet. Pour exécuter une opération α, il effectue la lecture sur cet état et diffuse le symbole d'entrée de l'opération α en utilisant la primitive de diffusion causale. Quand le processus p_j reçoit le message diffusé par p_i, il applique localement l'opération à son état courant.

Pour la mémoire, il est bien connu que cette stratégie implémente un peu plus que la mémoire [45]. Les situations causalement cohérentes qui ne peuvent pas être produites par l'algorithme de la figure 4.14 sont appelées *situations de fausse causalité*. La figure 4.13 explore cette nouvelle notion en proposant un nouveau critère de cohérence, la *cohérence causale forte*, qui admet exactement les histoires générées par l'algorithme de la figure 4.14. L'histoire de la figure 4.2e, détaillée sur la figure 4.13, est un exemple de fausse causalité. La proposition 4.5 prouve que la cohérence causale forte est le plus fort critère de cohérence implémenté par l'algorithme de la figure 4.14.

Proposition 4.5. *Soient un ADT $T \in \mathcal{T}$ et $H \in \mathcal{H}$ une histoire concurrente qui représente une exécution d'un système formé de processus communicants (c'est-à-dire que $\mathcal{P}_H$ forme une partition finie de E_H, chaque élément $p \in \mathcal{P}_H$ étant l'ensemble des événements de l'un des processus). H peut être générée par l'algorithme de la figure 4.14 si et seulement si $H \in SCC(T)$.*

Démonstration. Soient $T \in \mathcal{T}$ et $H \in \mathcal{H}$ telle que $\mathcal{P}_H$ forme une partition finie de E_H.

Le principe de l'algorithme de la figure 4.14 est de faire évoluer localement une variable locale en lui appliquant toutes les écritures quand un processus reçoit un message émis lors de leur appel. La séquentialité de chaque processus construit une linéarisation conforme à celle qui est exigée pour la cohérence pipeline correspondant à un ordre total $\dashrightarrow_p$. De plus, les linéarisations possibles sont restreintes par l'utilisation de la réception causale : si un processus p a déjà reçu un événement e quand il exécute e' (selon $\dashrightarrow_p$), ces deux événements sont dans le même ordre dans les linéarisations de tous les processus. Ainsi, une histoire concurrente H est admise par l'algorithme de la figure 4.14 si et seulement si pour chaque processus, il existe un bel ordre total $\dashrightarrow_p$ sur les événements de E_H qui contient l'ordre de processus et tel que :

- $\forall p \in \mathcal{P}_H, \text{lin}\left(H^{\dashrightarrow_p}[E_H/p]\right) \cap L(T) \neq \emptyset$,
- $\forall p, p' \in \mathcal{P}_H, \forall e \in E_H, \forall e' \in p, (e \dashrightarrow_p e') \Rightarrow (e \dashrightarrow_{p'} e')$.

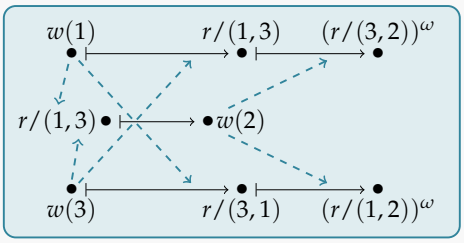
Supposons que H est admise par l'algorithme de la figure 4.14. On pose $\dashrightarrow = \left(\bigcap_{p \in \mathcal{P}_H} \dashrightarrow_p\right)$. Comme pour tout p $\dashrightarrow_p$ contient $\mapsto$, $\dashrightarrow$ contient également $\mapsto$. Soit $e \in E_H$. Comme $\dashrightarrow_p$ est un bel ordre, $\lfloor e \rfloor_{\dashrightarrow_p}$ est fini donc $\{e' \in E_H : e \not\dashrightarrow e'\} = \bigcup_{p \in \mathcal{P}_H}$, en tant qu'union finie d'ensembles finis, est fini et $\dashrightarrow$ est un ordre causal. Pour tout p et tout $e \in p$, on définit l_e comme le préfixe jusqu'à e de l'unique linéarisation de $\text{lin}\left(H^{\dashrightarrow_p}[E_H/p]\right) \cap L(T)$. Comme $\dashrightarrow \subset \dashrightarrow_p$, $l_e \in \text{lin}(H^{\dashrightarrow}[\lfloor e \rfloor_{\dashrightarrow}/\lfloor e \rfloor_{\mapsto}]) \cap L(T)$. Soient $e, e' \in E_H$ tels que $e \mapsto e'$. Il existe p tel que $\{e, e'\} \subset p$, donc l_e et $l_{e'}$ sont deux préfixes de l_i avec l_e moins long que $l_{e'}$, donc l_e est un suffixe de $l_{e'}$. Finalement, $H \in SCC(T)$.

Réciproquement, supposons que $H \in SCC(T)$. Notons l_e la linéarisation de chaque événement e définie par la cohérence causale forte. Reprenons la construction de la proposition 4.2. Soit $p \in \mathcal{P}_H$. On commence par définir $\dashrightarrow'_p$ par, pour tout $e, e' \in E_H$, $e \dashrightarrow'_p e'$ si $e \dashrightarrow e'$ ou s'il existe $e'' \in p$ tel que, dans la linéarisation $l_{e''}$, e est placé avant

COHÉRENCE CAUSALE FORTE

✗ Non composable p. 41
✓ Décomposable p. 41
✓ Faible p. 126

La cohérence causale forte est le critère que l'on obtient quand tous les processus exécutent les écritures des autres processus dans un ordre compatible avec l'ordre causal. La différence avec la cohérence causale est que les linéarisations successives des événements qui ont lieu sur le même processus sont des préfixes les unes des autres. L'histoire de la figure 4.2f (rappelée ci-contre) est fortement causalement cohérente : par exemple, le premier processus écrit 1 et passe dans l'état $(0,1)$, puis exécute l'écriture du 3 et passe dans l'état $(1,3)$ qu'il lit, puis exécute l'écriture du 2 et termine dans l'état $(3,2)$. Les autres processus se comportent de manière similaire.



Formellement, une histoire H est fortement causalement cohérente par rapport à un type de données abstrait T (définition 4.5) s'il existe un ordre causal $\dashrightarrow$ et une séquence l_e pour chaque événement e de H tels que :

- *pour tout e, l_e est une linéarisation de l'histoire $H^{\dashrightarrow}[\lfloor e \rfloor_{\dashrightarrow} / \lfloor e \rfloor_{\mapsto}]$ composée des lectures du passé local et des écritures du passé causal de e, triées selon l'ordre causal (c'est la propriété demandée pour la cohérence causale) ;*
- *les linéarisations des événements d'un processus sont préfixes l'une de l'autre, c'est à dire pour toute paire d'événements ordonnés selon l'ordre de processus $e \mapsto e'$, l_e est un préfixe de $l_{e'}$.*

Dans l'exemple ci-dessus, les linéarisations des trois premières opérations du premier processus sont :

$$w(1)/\bot \qquad w(1)/\bot \cdot w(3) \cdot r \cdot r/(1,3) \qquad w(1)/\bot \cdot w(3) \cdot r \cdot r/(1,3) \cdot w(2) \cdot r/(3,2)$$

Définition 4.5 (Cohérence causale forte). La *cohérence causale forte* est le critère de cohérence

$$SCC : \left\{ \begin{array}{lll} \mathcal{T} & \to & \mathcal{P}(\mathcal{H}) \\ T & \mapsto & \left\{ H \in \mathcal{H} : \begin{array}{l} \exists \dashrightarrow \in co(H), \exists (l_e)_{e \in E_H}, \\ \quad \forall e \in E_H, l_e \in \mathrm{lin}(H^{\dashrightarrow}[\lfloor e \rfloor_{\dashrightarrow} / \lfloor e \rfloor_{\mapsto}]) \cap L(T) \\ \wedge \quad \forall e, e' \in E_H, e \mapsto e' \Rightarrow \exists l' \in \Sigma^\star, l_{e'} = l_e \cdot l' \end{array} \right\} \end{array} \right.$$

L'histoire de la figure 4.2e (à droite) est causalement cohérente mais pas fortement causalement cohérente. En effet, en ignorant les lectures cachées, la linéarisation pour la première lecture du premier processus ne peut être que $w(3) \cdot w(1)/\bot \cdot r/(3,1)$. Il en résulte que celle de la première écriture du premier processus est $w(3) \cdot w(1)/\bot$, donc l'écriture du 3 précède celle du 1 dans l'ordre causal. Or la situation est symétrique avec le troisième processus, d'où un cycle dans l'ordre causal.

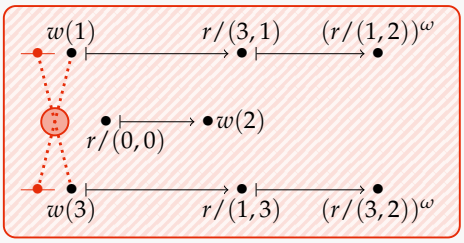


FIGURE 4.13 – La cohérence causale forte.

e' ou e est présent mais pas e'. La relation $e \dashrightarrow'_p e'$ est bien une relation d'ordre car les linéarisations des événements de p sont liées par une relation de préfixation (les événements sont placés dans le même ordre) et elles respectent l'ordre causal. On étend $\dashrightarrow'_p$ en un ordre total $\dashrightarrow_p$. Celui-ci contient $\mapsto$ et par construction, la seule linéarisation l_p de $\mathrm{lin}(H^{\dashrightarrow_p}[E_H/p])$ contient l_e pour tout $e \in p$, donc $l_p \in L(T)$. Soient $p, p' \in \mathcal{P}_H$, $e \in E_H$ et $e' \in p$ tels que $e \dashrightarrow_p e'$. Comme e' apparaît dans sa propre linéarisation, cela signifie que e fait partie de la linéarisation de e', donc $e \dashrightarrow e'$. Il en résulte que $e \dashrightarrow'_{p'} e'$ donc $e \dashrightarrow_{p'} e'$. Finalement, H est admise par l'algorithme de la figure 4.14. □

```
1  algorithm SCC(A, B, Z, ζ0, τ, δ)
2  |  variable state_i ∈ Z ← ζ0 ;                          // état local
3  |  operation apply (α ∈ A) ∈ B
4  |  |  variable β ∈ B ← δ(state_i, α) ;                    // lecture
5  |  |  causal broadcast message (α) ;                     // écriture
6  |  |  return β ;
7  |  end
8  |  on receive message (α ∈ A)
9  |  |  state_i ← τ(state'_i, α') ;
10 |  end
11 end
```

FIGURE 4.14 – Algorithme générique $SCC(T)$: code pour p_i

4.3 Comportements particuliers

L'ordre total qui décrit la linéarisation de la cohérence séquentielle est un ordre causal : il contient l'ordre des processus et tout événement a un futur causal cofini puisque sa position (finie) dans la linéarisation détermine son passé causal. De plus, comme l'ordre est total, le présent concurrent de tous les événements est vide. La réciproque est vraie également : une histoire faiblement causalement cohérente dans laquelle chaque événement a un présent concurrent vide est séquentiellement cohérente.

Cela signifie qu'il n'est pas nécessaire de gérer plusieurs fois la synchronisation dans un même programme. Plus précisément, les histoires générées par une application répartie utilisant un objet partagé dans son implémentation sont restreintes à la fois par la spécification de l'objet partagé (sa spécification séquentielle et son critère de cohérence) et par l'application elle-même (par exemple si une opération de l'objet n'est jamais appelée dans l'implémentation de l'application, cette opération ne sera pas présente dans les histoires générées). Si un mécanisme du programme garantit que deux opérations différentes sont toujours causalement dépendantes, un objet faiblement causalement cohérent se comportera comme sa contrepartie séquentiellement cohérente. Imaginons par exemple un jeu au tour par tour comme le jeu d'échecs. D'après les règles du jeu (le programme), chaque joueur (un processus) doit attendre que l'autre joueur ait terminé son tour (dépendance causale entre le déplacement de deux pièces, soit deux écritures) pour pouvoir déplacer une pièce sur le plateau (l'objet partagé). Utiliser la cohérence causale faible plutôt que la cohérence séquentielle pour implémenter le plateau ne changera en rien le comportement du programme.

En fait, dans l'exemple précédent, l'ordre causal n'est total que si une pendule d'échecs sonne entre chaque tour (mécanisme externe de synchronisation). Sinon, le joueur dont ce n'est pas le tour de déplacer une pièce (écriture) doit regarder le plateau (lectures) pour savoir quand il peut jouer. L'article original sur la mémoire causale [6] justifie l'intérêt de celle-ci par le fait que toute histoire dans laquelle deux écritures sont toujours causalement ordonnées est séquentiellement cohérente. La proposition 4.6 prouve que cela est en fait une propriété de la cohérence causale faible : toute histoire faiblement causalement cohérente (WCC) dans laquelle toutes les écritures sont ordonnées deux à deux est séquentiellement cohérente (SC).

Proposition 4.6. *Soient un ADT T et une histoire concurrente $H \in WCC(T)$. Si pour tout couple d'écritures $(u, u') \in U^2_{T,H}$, $u \dashrightarrow u'$ ou $u' \dashrightarrow u$, alors $H \in SC(T)$.*

Démonstration. Soient $T \in \mathcal{T}$ et $H \in WCC(T)$ telle que pour tout couple d'écritures $(u, u') \in U^2_{T,H}$, $u \dashrightarrow u'$ ou $u' \dashrightarrow u$.

Soit $\leq$ un ordre total sur E_H qui étend $\dashrightarrow$, et soit l l'unique linéarisation de $\text{lin}(H^{\leq})$. Comme $\mapsto \subset \dashrightarrow \subset \leq$, $l \in \text{lin}(H)$. Supposons que $l \notin L(T)$. Comme le système de transitions de T est déterministe, il existe un préfixe fini non vide de l qui n'appartient pas à $L(T)$. Soient $l' \in \Sigma^\star$ et $e \in E_H$ tels que $l' \cdot \Lambda(e)$ soit le plus court tel préfixe. Comme $H \in WCC(T)$, il existe une linéarisation dans $\text{lin}(H^{\dashrightarrow}[\lfloor e \rfloor_{\dashrightarrow} / \{e\}]) \cap L(T)$ de la forme $l'' \cdot \Lambda(e)$ puisque e est le maximum de $\lfloor e \rfloor_{\dashrightarrow}$ selon $\dashrightarrow$. Or l' et l'' possèdent les mêmes écritures ordonnées dans le même ordre, car $\dashrightarrow$ est total sur les écritures, donc l' et l'' mènent dans le même état. Dans ces conditions, il est absurde d'avoir $l'' \cdot \Lambda(e) \in L(T)$, $l' \in L(T)$ et $l' \cdot \Lambda(e) \notin L(T)$. Donc $l \in L(T)$ et $H \in SC(T)$. □

Une façon simple (bien que peu tolérante aux pannes) d'utiliser cette propriété est de se servir d'un jeton de synchronisation. Un programme implémenté selon cette stratégie utilise, en plus d'un objet partagé représentant ses données, un jeton dont la valeur désigne en permanence l'identifiant d'un processus. Seul le processus désigné par le jeton est autorisé à écrire sur le premier objet. Quand un processus a terminé ses écritures, il désigne un autre processus en écrivant son identifiant dans le jeton. Une telle stratégie garantit que deux écritures sont toujours ordonnées causalement. La proposition 4.6 permet donc d'affirmer que la cohérence causale faible, appliquée à la composition du premier objet et du jeton, offrira exactement la même qualité de service que la cohérence séquentielle dans cette situation, mais pour un coût beaucoup plus faible.

La proposition s'applique également aux autres critères causaux discutés dans ce chapitre, plus forts que la cohérence causale faible. De plus, la proposition 4.7 prouve que les histoires causalement convergentes dans lesquelles il n'y a pas d'écritures incomparables selon l'ordre causal à une lecture sont également séquentiellement cohérentes. La convergence causale renforce la cohérence d'écritures, dans laquelle certaines lectures peuvent sembler incorrectes. Cette proposition affirme que les seules lectures qui peuvent sembler incorrectes dans la convergence causale sont celles qui sont incomparables avec au moins une écriture selon l'ordre causal.

Proposition 4.7. *Soient un ADT T et une histoire concurrente $H \in CCv(T)$ tels que pour tout $u \in U_{T,H}$ et $q \in Q_{T,H}$, $u \dashrightarrow q$ ou $q \dashrightarrow u$. Alors $H \in SC(T)$.*

Démonstration. Soient $T \in \mathcal{T}$ et $H \in CCv(T)$ tels que pour tout $u \in U_{T,H}$ et $q \in Q_{T,H}$, $u \dashrightarrow q$ ou $q \dashrightarrow u$. Comme $H \in CCv(T)$, il existe un ordre total $\leq$ qui contient $\dashrightarrow$ et, pour tout $e \in E_H$, une linéarisation $l_e \cdot \Lambda(e) \in \text{lin}(H^{\leq}[\lfloor e \rfloor_{\dashrightarrow} / \{e\}] \cap L(T)$.

Soit l l'unique linéarisation de $\text{lin}(H^{\leq})$. Comme $\mapsto \subset \leq$, $l \in \text{lin}(H)$. Supposons que $l \notin L(T)$. Comme dans la proposition 4.6, l possède un préfixe $l' \cdot \Lambda(e) \notin L(T)$ avec $l' \in L(T)$. L'événement e ne peut pas être étiqueté par une écriture pure car, le système de transitions de T étant complet, toute écriture pure peut avoir lieu dans tout état donc en particulier dans l'état obtenu après exécution de l'. L'événement e ne peut alors être qu'une lecture, donc par hypothèse, elle est causalement ordonnée avec toutes les écritures de l'histoire. Comme l'ordre total $\leq$ respecte l'ordre de causal, $\dashrightarrow$, les écritures du passé de e selon l'ordre causal et l'ordre total sont exactement les mêmes : $\lfloor e \rfloor_{\dashrightarrow} \cap U_{T,H} = \lfloor e \rfloor_{\leq} \cap U_{T,H}$. On en déduit que l_e et l' possèdent les mêmes écritures dans le même ordre, donc il est impossible que $l_e \cdot \Lambda(e) \in L(T)$, $l' \in L(T)$ et $l' \cdot \Lambda(e) \notin L(T)$. Finalement, $l \in L(T)$ et $H \in SC(T)$. □

Conclusion

Dans ce chapitre, nous avons étudié la causalité dans les critères de cohérence. Nous avons étendu la notion de mémoire causale à tous les types de données abstraits en définissant la cohérence causale comme un critère de cohérence. Nous avons également exploré des variantes de la cohérence causale autour de quatre critères de cohérence.

Ces quatre critères sont pertinents. La *cohérence causale faible* peut être vue comme le dénominateur causal des trois autres critères causaux. La *convergence causale* est le résultat de la conjonction de la cohérence causale faible et de la cohérence d'écritures. La *cohérence causale*, qui renforce la cohérence causale faible et la cohérence pipeline, est une généralisation de la mémoire causale à tous les types de données abstraits. Enfin, tous les types de données abstraits possèdent une implémentation respectant chacun de ces critères dans les systèmes sans-attente. La *cohérence causale forte* modélise exactement l'implémentation standard de la cohérence causale par passage de messages.

Finalement, ce chapitre permet une compréhension plus fine de ce qu'apporte la causalité dans les critères de cohérence. Les deux critères à garder en mémoire sont la cohérence causale et la convergence causale qui renforcent respectivement la cohérence pipeline et la cohérence d'écritures.

Une question qui se pose est celle de l'implémentation efficace de la convergence causale. La convergence causale correspond à la cohérence d'écritures forte dans laquelle la relation de visibilité est transitive (d'où la similarité entre les preuves des algorithmes des figures 4.14 et 3.7). Or la cohérence d'écritures forte est plus coûteuse à implémenter que la cohérence d'écritures. En particulier, seul le premier des trois algorithmes exposés dans le chapitre 3 implémente la cohérence d'écritures forte, et peut donc être adapté pour implémenter la convergence causale. Cependant, les propriétés apportées par la cohérence d'écritures forte mais pas par la cohérence d'écritures ne sont pas forcément nécessaires avec la cohérence causale faible, qui spécifie également les lectures faites avant convergence. Lorsque cela est possible, il peut être moins coûteux et tout aussi puissant d'utiliser la conjonction de la cohérence causale faible et de la cohérence d'écritures ($WCC + UC$) plutôt que la convergence causale. Il serait intéressant d'étudier des algorithmes plus performants pour implémenter la conjonction de la cohérence causale faible et de la cohérence d'écritures.

5

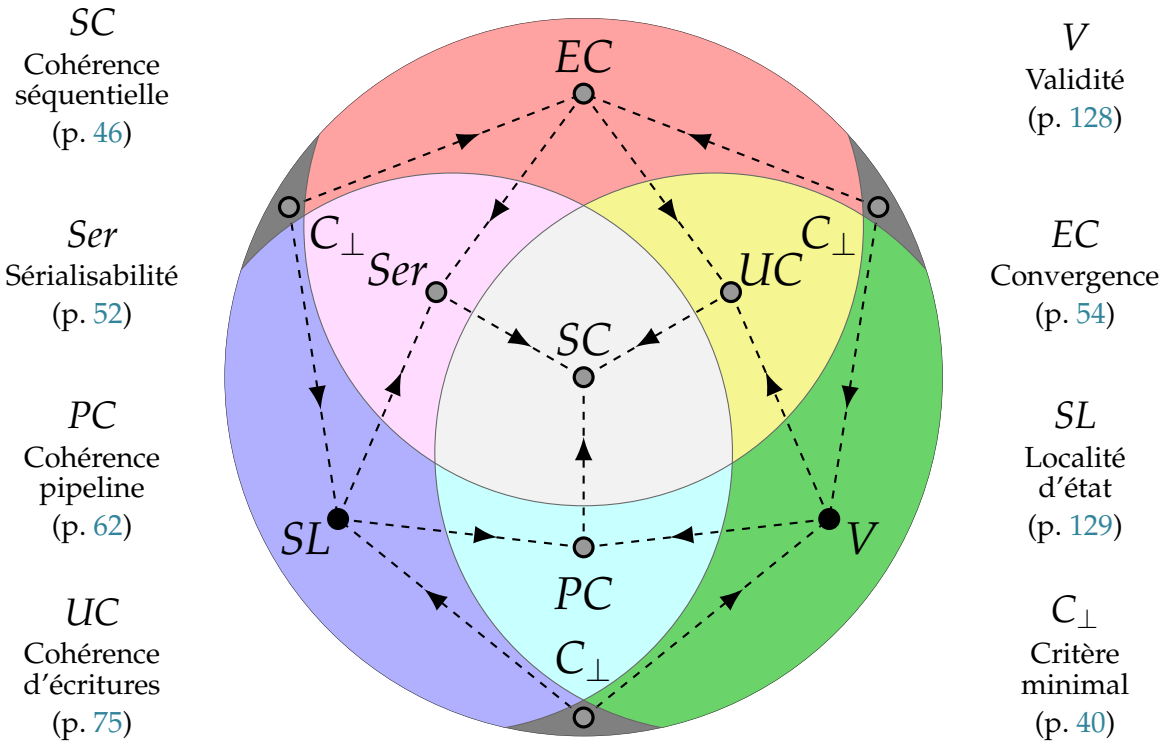
L'espace des critères faibles

Sommaire

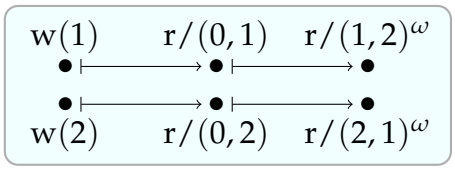
Introduction

Dans ce chapitre, nous nous intéressons spécifiquement aux objets partagés que l'on peut implémenter dans les systèmes répartis asynchrones à passage de messages sans-attente (ou simplement « systèmes sans-attente », $AS_n[\emptyset]$, voir section 2.1.2). Selon le théorème CAP [26], il n'existe pas d'algorithme qui assure les trois propriétés suivantes à la fois.

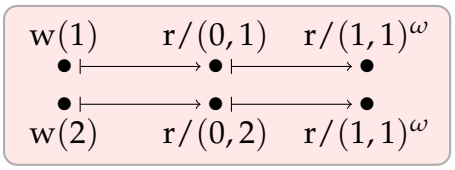
Cohérence forte. Dans leur preuve du théorème CAP [48], Gilbert et Lynch définissent la cohérence forte par un registre atomique (c'est-à-dire linéarisable). Le résultat peut être étendu à la mémoire séquentiellement cohérente, à condition de considérer au moins deux registres : dans [14], Attiya et Welch prouvent que pour implémenter la mémoire séquentiellement cohérente dans un système synchrone,



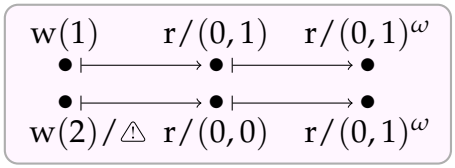
(a) Structure de l'espace des critères faibles.



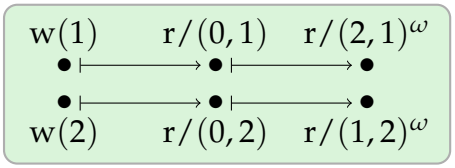
(b) Cohérence pipeline



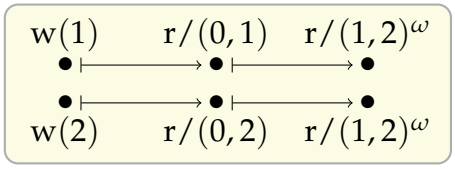
(c) Convergence



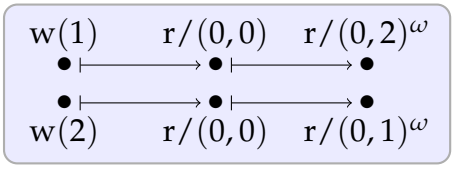
(d) Sérialisabilité



(e) Validité



(f) Cohérence d'écritures



(g) Localité d'état

FIGURE 5.1 – Sur ces six histoires concurrentes, deux processus partagent un flux fenêtré de taille 2 (voir page 29). Le premier processus écrit 1 dans le flux fenêtré (opération $w(1)$) puis lit indéfiniment (opération r). Le deuxième processus a un rôle similaire mais il écrit 2 initialement.

même sans considérer les pannes, le temps d'exécution des opérations d'écriture ou des opérations de lecture doit être au moins proportionnel au délai de transmission d'un message. Le résultat reste vrai dans un système sans-attente, qui suppose moins d'hypothèses.

Disponibilité (« Availability »). La disponibilité impose que les appels aux méthodes d'un objet finissent toujours par retourner une réponse.

Résistance au Partitionnement. Un système *partitionable* est soumis à des *partitionnements* temporaires ou définitifs, pendant lesquelles deux processus dans des *partitions* différentes sont incapables de communiquer entre eux. Dans la preuve du théorème CAP [48], les partitionnements sont modélisés par des pertes de messages entre les processus de partitions différentes. Si un processus est seul dans sa partition, il lui est impossible d'attendre des messages d'autres processus. Les systèmes sans-attente sont donc également une bonne modélisation des systèmes partitionables.

Au regard de cette discussion, nous exprimons le théorème CAP par le théorème 5.1 : il est impossible d'implémenter une mémoire séquentiellement cohérente composée d'au moins deux registres dans un système réparti asynchrone où l'on ne fait pas l'hypothèse d'une majorité de processus corrects. Le résultat reste vrai pour les systèmes sans-attente. Nous ne donnons pas ici la preuve de ce théorème, qui est très similaire à la fin de celle du théorème 5.7.

Théorème 5.1 (Théorème CAP)**.** *Soient x et y deux noms de registre, $n > 1$ processus et $f \geq \frac{n}{2}$. Il n'existe pas d'algorithme implémentant SC $\left(\mathcal{M}_{\{x,y\}}\right)$ dans $AS_n[t \leq f]$.*

Problème. *Quels objets partagés peuvent être implémentés dans les systèmes sans-attente ?*

La première question que nous nous posons est celle des critères de cohérence pour lesquels il est possible d'implémenter *tous* les types de données abstraits (ADT) dans un système sans-attente. Un tel critère de cohérence est appelé un *critère faible*, par opposition aux *critères forts* comme la cohérence séquentielle et la linéarisabilité, soumis au théorème CAP. Le but de ce chapitre est d'étudier l'espace des critères faibles globalement. Pour cela, nous poursuivons la longue quête du plus fort des critères faibles.

Notre deuxième question est celle des objets que l'on peut implémenter dans les systèmes sans-attente et vérifiant un critère fort donné. Cela nous permet de mieux comprendre les propriétés des types de données abstraits qui permettent l'implémentation de critères plus forts.

Approche. *Nous cherchons des couples de critères faibles dont la conjonction est un critère fort. Pour prouver qu'un critère de cohérence est fort, nous adaptons les outils développés en algorithmique du réparti aux systèmes sans-attente.*

Nous identifions trois critères faibles, dits *primaires* qui peuvent être conjugués deux à deux pour former des critères *secondaires* faibles, mais dont la conjonction forme un critère fort. Les trois critères secondaires obtenus sont la cohérence pipeline, la cohérence d'écritures et la sérialisabilité. Cela permet de justifier la structure visible sur la figure 5.1a.

Contribution. *Ce chapitre présente quatre contributions principales.*

- *Nous montrons qu'il n'existe pas de plus fort critère de cohérence faible.*
- *Nous introduisons deux nouveaux critères de cohérence qui justifient une structuration de l'espace des critères faibles autour de trois critères primaires et de trois critères secondaires, comme montré sur la figure 5.1a.*
- *Nous prouvons que le nombre de Consensus du flux fenêtré de taille k, avec la cohérence séquentielle et avec la conjonction des trois critères faibles, est égal à k, ce qui remplit élégamment la hiérarchie de Herlihy en exhibant un objet partagé intelligible pour chaque échelon de la hiérarchie. De plus, à notre connaissance, c'est la première fois qu'est identifié un objet partagé vérifiant un critère de cohérence strictement plus faible que la cohérence séquentielle et ayant un nombre de Consensus différent de* 1.
- *Nous montrons l'existence d'une hiérarchie plus fine pour les types de données abstraits pour lesquels l'application de la cohérence séquentielle a un nombre de Consensus nul, en faisant varier le critère de cohérence.*

Le reste de ce chapitre est organisé comme suit. La partie 5.1 définit ce qu'est un critère de cohérence faible et introduit deux nouveaux critères de cohérence : la validité et la localité d'état. La partie 5.2 étudie la structure de l'espace des critères faibles en s'intéressant aux flux fenêtrés. La partie 5.3 étudie les cas particuliers de la mémoire et des types de données abstraits dans lesquels les écritures commutent deux à deux. Finalement, la partie 5.4 discute des cas où chaque critère est le plus pertinent.

5.1 Critères faibles

5.1.1 Définitions

Avant de discuter de l'existence du plus fort des critères faibles, il convient de fixer une définition des critères faibles. Dans le sens le plus général, les critères faibles sont définis par opposition aux critères forts tels que la cohérence séquentielle et la linéarisabilité. Un critère faible est donc simplement un critère qui ne cache pas complètement la concurrence entre les processus. Cette définition est trop vague pour être utilisée de manière formelle.

Le résultat principal concernant la cohérence forte et la justification principale de l'intérêt donné aux critères faibles est le théorème CAP qui énonce l'impossibilité d'implémenter les critères forts dans les systèmes partitionables, et donc dans $AS_n[\emptyset]$. Nous proposons de définir les critères faibles en négatif, comme un critère que l'on peut implémenter dans ce système.

En pratique, un programmeur choisira en premier le type de données abstrait (ADT) qu'il veut utiliser en fonction de l'application qu'il veut lui donner puis, devant l'absence d'implémentations ou le coût d'un critère fort, il choisira les propriétés qu'il accepte de perdre. Utiliser un ADT discriminatoire particulier pour définir « implémentable » est dangereux car il ne dit potentiellement rien sur les autres ADT : l'objet Consensus a été prouvé universel pour la linéarisabilité mais rien ne dit qu'il l'est également pour tous les autres critères. Pour dire qu'un critère peut être implémenté dans $AS_n[\emptyset]$, il est donc nécessaire d'exhiber un algorithme pour chaque ADT.

On veut également restreindre l'analyse aux objets effectivement partagés. Par exemple, la cohérence locale, qui peut être implémentée pour n'importe quel type même dans des systèmes qui ne permettent aucune communication, ne nous intéresse pas dans cette étude. On définit donc les critères partagés (définition 5.1) comme ceux qui sont comparables à la cohérence séquentielle (SC). L'espace des critères partagés

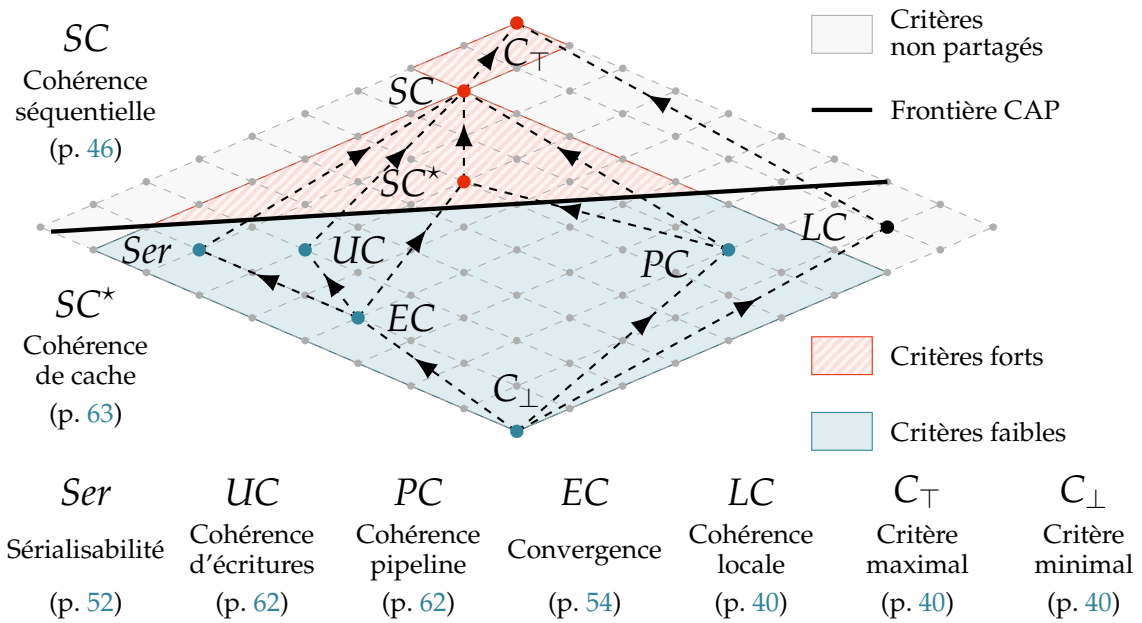


FIGURE 5.2 – Treillis des critères de cohérence.

est partitionné entre celui des critères faibles pour lesquels il existe un algorithme qui implémente chaque type de données abstrait dans les systèmes sans-attente (définition 5.2) et celui des critères forts (définition 5.3).

Définition 5.1 (Critère partagé)**.** Un critère C est *partagé* si $C \leq SC$ ou $SC \leq C$.

Définition 5.2 (Critère faible)**.** Un critère de cohérence partagé C est *faible* si, pour tout type de données abstrait T, il existe un algorithme A tel que, quand il est exécuté dans le système $AS_n[\emptyset]$,

- l'algorithme A permet d'appeler toutes les opérations de T ;
- toutes les actions (appels d'opérations et réceptions de messages) terminent ;
- toutes les histoires admises par A font partie de $C(T)$.

L'espace des critères faibles est noté $\mathcal{C}_W$.

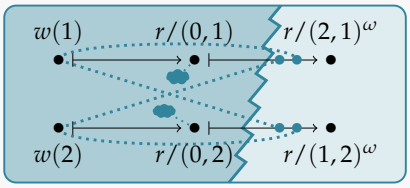
Définition 5.3 (Critère fort)**.** Un critère de cohérence partagé est *fort* s'il n'est pas faible. L'espace des critères forts est noté $\mathcal{C}_S$.

D'après ces définitions, l'existence d'un plus fort critère faible semble peu probable. En effet, comme représenté sur la figure 5.2, l'ensemble des critères partagés forme un cône de sommet SC. Or, la frontière du théorème CAP qui sépare les critères forts des critères faibles coupe ce cône en dessous du sommet donc $\mathcal{C}_W$ a une forme de cône tronqué. Pour qu'il existe un maximum dans $\mathcal{C}_W$, il faudrait soit que le cône des critères partagés soit réduit à une droite, c'est-à-dire que les critères partagés soient totalement ordonnés (au moins localement juste en dessous de SC), soit que la frontière CAP soit « parallèle à l'un des bords du cône », c'est-à-dire que l'on puisse identifier une (ou plusieurs) propriété élémentaire de la cohérence séquentielle portant le théorème CAP, un critère partagé étant faible si et seulement si il ne garantit pas cette propriété.

VALIDITÉ

✗ Non composable p. 41
✓ Décomposable p. 41
✓ Faible p. 126

La validité garantit que, si tous les processus arrêtent d'écrire, alors, au bout d'un moment, toutes les valeurs lues seront compatibles avec une linéarisation des écritures de l'histoire. Par exemple, dans l'histoire de la figure 5.1e (rappelée ci-contre), le premier processus finit par lire l'état $(2,1)$, qui correspond à l'exécution de $w(2)$ suivi de celle de $w(1)$. Le deuxième processus se stabilise lui dans l'état $(1,2)$ qui correspond à l'exécution des deux écritures dans l'autre sens.



Formellement, une histoire H est valide par rapport à un type de données abstrait T (définition 5.4) si l'une des deux propriétés suivante est vérifiée.

- *H contient une infinité d'écritures ($|U_{T,H}| = \infty$)*
- *il existe un ensemble cofini E' d'événements tels que pour chacun d'entre eux, il existe une linéarisation correcte par rapport à la spécification séquentielle de T, de l'histoire $H[E_H/\{e\}]$ qui contient toutes les écritures de H, ainsi que l'événement e considéré.*

Dans l'exemple ci-dessus, l'ensemble E' contient les lectures qui retournent $(2,1)$ ou $(1,2)$. Les linéarisations demandées pour la première lecture de chaque processus est :

$$w(2)\cdot r\cdot w(1)\cdot r/(2,1) \qquad w(1)\cdot r\cdot w(2)\cdot r/(1,2)$$

Définition 5.4 (Validité). La validité est le critère de cohérence :

$$V : \left\{ \begin{array}{ccc} \mathcal{T} & \to & \mathcal{P}(\mathcal{H}) \\ T & \mapsto & \left\{ H \in \mathcal{H} : \begin{array}{l} |U_{T,H}| = \infty \\ \vee \quad \exists E' \subset E_H, (|E_H \setminus E'| < \infty \\ \qquad \wedge \quad \forall e \in E', \mathrm{lin}\,(H[E_H/\{e\}]) \cap L(T) \neq \emptyset) \end{array} \right\} \end{array} \right.$$

L'histoire de la figure 5.1d (ci-contre) n'est pas valide. En effet, l'infinité de lectures du deuxième processus se situe après l'écriture $w(2)$. De plus, une infinité de ces écritures doit être présente dans E'. Il n'est donc pas possible qu'elles ne retournent pas la valeur 2.

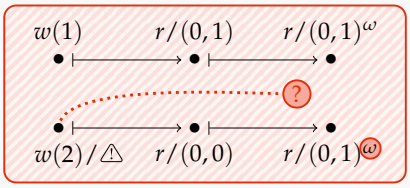


FIGURE 5.3 – La validité.

5.1.2 Validité et localité d'état

Dans toutes les histoires de la figure 5.1, deux processus partagent une instance de flux fenêtré de taille 2. Le premier processus appelle l'écriture $\mathrm{w}(1)$ puis lit une infinité de fois et le deuxième processus appelle l'écriture $\mathrm{w}(2)$ puis lit une infinité de fois. La plupart des critères étudiés dans cette thèse imposent que la suite des valeurs lues par chaque processus soit constante à partir d'un certain point, dit de *stabilisation*. Dans ces histoires, la stabilisation est atteinte dès la deuxième lecture.

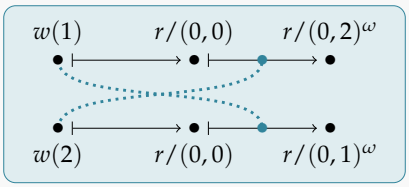
Supposons dans un premier temps que l'objet partagé est séquentiellement cohérent. Les linéarisations font partie du langage $\mathrm{w}(i)/\perp \cdot \mathrm{r}/(0,i)^\star \cdot \mathrm{w}(j)/\perp \cdot \mathrm{r}/(i,j)^\omega$, avec $i \neq j$ et $\{i,j\} = \{1,2\}$. En particulier, les trois propriétés suivantes sont vraies.

Validité. La valeur lue par chaque processus après stabilisation vaut soit $(1,2)$, soit $(2,1)$ (figure 5.1e). En d'autres termes, toute valeur lue après stabilisation est

LOCALITÉ D'ÉTAT

✗ Non composable p. 41
✓ Décomposable p. 41
✓ Faible p. 126

La localité d'état modélise le fait que l'état sur un processus ne peut pas changer spontanément sans exécuter une écriture de l'histoire. Cela se traduit par le fait que pour chaque processus, il existe une séquence d'opérations contenant toutes les lectures pures du processus et une partie des écritures de l'histoire, correcte vis-à-vis de la spécification séquentielle de l'objet : les changements d'état observés entre deux lectures ne peuvent donc venir que de l'exécution des écritures situées entre les deux lectures dans cette linéarisation. Par exemple, dans l'histoire de la figure 5.1g (rappelée ci-contre), le premier processus lit initialement la valeur $(0,0)$ puis la valeur $(0,2)$. Cela peut être interprété comme un passage de l'état local $(0,0)$ à l'état local $(0,2)$ lors de l'exécution de l'écriture $w(2)$ faite par le deuxième processus.



Formellement, une histoire H vérifie la localité d'état par rapport à un type de données abstrait T (définition 5.5) si, pour tout processus $p \in \mathcal{P}_H$, il existe un ensemble d'événements C_p qui contient toutes les lectures pures (la projection ne conservant que celles de p), tel que l'histoire $H[p \cap C_p / C_p]$ formée des lectures de p et des écritures de C_p possède une linéarisation compatible avec la spécification séquentielle de T. Dans l'exemple ci-dessus, l'ensemble C_p contient les lectures pures et, pour chaque processus, l'écriture de l'autre processus. Les deux linéarisations suivantes correspondent à la propriété pour les deux processus :

$$r/(0,0) \cdot w(2) \cdot (r \cdot r/(0,2))^\omega \qquad r/(0,0) \cdot w(1) \cdot (r \cdot r/(0,1))^\omega$$

Définition 5.5 (Localité d'état). La *localité d'état* est le critère de cohérence :

$$SL : \left\{ \begin{array}{lll} \mathcal{T} & \to & \mathcal{P}(\mathcal{H}) \\ T & \mapsto & \left\{ H \in \mathcal{H} : \begin{array}{rl} \forall p \in \mathcal{P}_H, \exists C_p \subset E_H, & \hat{Q}_{T,H} \subset C_p \\ \wedge & \mathrm{lin}(H[p \cap C_p / C_p]) \cap L(T) \neq \emptyset \end{array} \right\} \end{array} \right.$$

L'histoire de la figure 5.1f (à droite) ne vérifie pas la localité d'état. En effet, pour que le deuxième processus passe de l'état $(0,2)$ à l'état $(1,2)$, il doit exécuter une écriture $w(1)$ puis une écriture $w(2)$. L'histoire ne contient qu'une seule écriture $w(2)$ que le processus a déjà exécutée pour passer de l'état $(0,0)$ à l'état $(0,2)$. Il est donc impossible de construire la linéarisation souhaitée pour le deuxième processus.

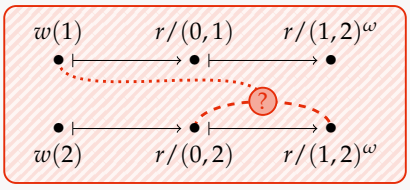


FIGURE 5.4 – La localité d'état.

le résultat d'une linéarisation de toutes les écritures de l'histoire, qui respecte l'ordre de processus. La validité est formalisée sous la forme d'un critère de cohérence sur la figure 5.3.

Convergence. Les valeurs lues par tous les processus après stabilisation sont identiques (figure 5.1c). Cette propriété correspond exactement au critère de cohérence du même nom (voir page 54).

Localité d'état. Les seules lectures successives autorisées pour un processus sont $(0,1)$ suivie de $(1,2)$ ou bien $(0,2)$ suivie de $(2,1)$ (figure 5.1g). Par contre, il n'est pas possible de lire $(1,2)$ suivi de $(2,1)$ par exemple. Autrement dit, chaque processus donne l'impression de passer de l'état $(0,0)$ à l'état $(0,1)$ puis à l'état $(1,2)$, ou bien de l'état $(0,0)$ à l'état $(0,2)$ puis à l'état $(2,1)$. La localité d'état,

```
1 algorithm Consensus
2   variable flux_i : W_k ;                          // flux_i vérifie V + EC + SL
3   operation propose (v ∈ ℕ) ∈ ℕ
4     flux_i.w(v + 1) ;
5     variable tuple : ℕ^k ← flux_i.r() ;
6     return tuple[min{0 ≤ j < k : tuple[j] ≠ 0}] − 1 ;
7   end
8 end
```

FIGURE 5.5 – Du flux fenêtré $(V + EC + SL)$-cohérent au Consensus : code pour p_i.

formalisée sous la forme d'un critère de cohérence sur la figure 5.4, modélise le fait que chaque processus se comporte comme s'il possédait un état local qui n'évoluait que grâce à l'exécution locale des écritures présentes dans l'histoire.

Nous pouvons maintenant trier les critères de cohérence définis dans les chapitres précédents en fonction des propriétés qu'ils vérifient.

Convergence, validité ou localité d'état seule. La cohérence causale faible est un critère faible plus fort que la validité. La convergence forte est un critère faible plus fort que la convergence.

Validité et localité d'état. La cohérence pipeline (figure 5.1f), la cohérence causale et la cohérence causale forte renforcent à la fois la validité et la localité d'état. De plus, tous les trois sont des critères faibles puisqu'ils sont plus faibles que la cohérence causale forte qui peut être implémentée d'après le chapitre 4.

Convergence et localité d'état. La sérialisabilité (figure 5.1d) est à la fois plus forte que la convergence (dans la version retenue dans cette thèse, où les lectures pures ne peuvent pas avorter) et que la localité d'état. Il s'agit également d'un critère faible puisqu'il est possible de faire avorter toutes les écritures, sans même communiquer.

Convergence et validité. La cohérence d'écritures (figure 5.1f), la cohérence d'écritures forte et la convergence causale renforcent à la fois la convergence et la validité. Ce sont bien des critères faibles d'après l'algorithme de la convergence causale présenté dans le chapitre 4.

Convergence, validité et localité d'état. Dans tous les critères présentés dans les chapitres précédents, seule la cohérence séquentielle (et $C_\top$) vérifie à la fois les trois propriétés, mais il s'agit d'un critère fort d'après le théorème CAP.

5.2 Structure de l'espace des critères faibles

5.2.1 Critères primaires et secondaires

La validité (V), la convergence (EC) et la localité d'état (SL) sont tous les trois des critères faibles, puisqu'ils sont tous plus faibles soit que la convergence causale (validité et convergence) soit que la cohérence causale (validité et localité d'état) pour lesquels nous avons donné des algorithmes dans le chapitre 4. Nous prouvons maintenant que la conjonction de ces trois critères $(V + EC + SL)$ est un critère fort. Pour

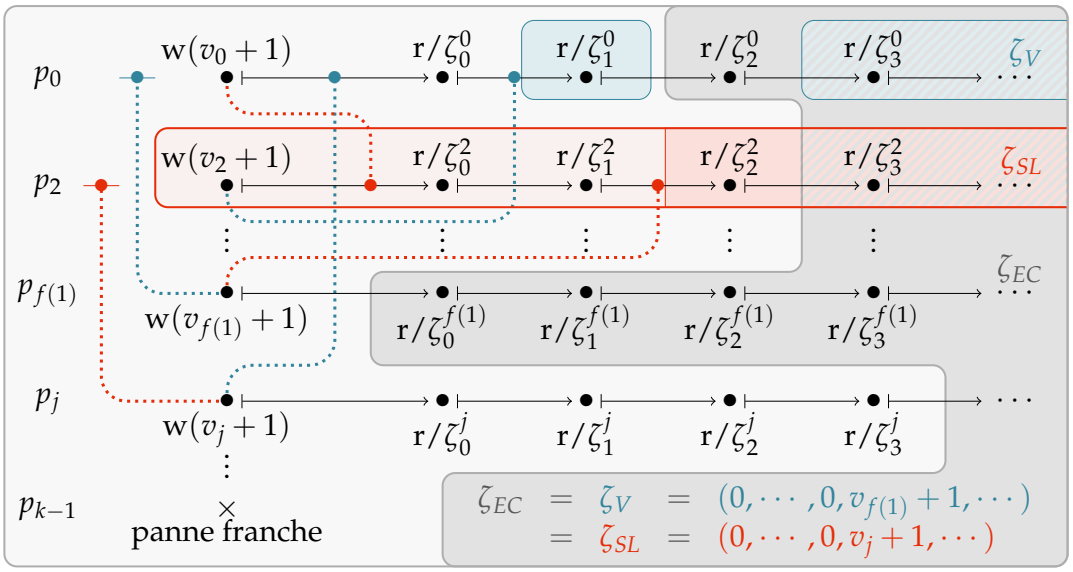


FIGURE 5.6 – Extension de l'histoire produite par l'algorithme de la figure 5.5. Les informations apportées par la convergence sont en gris, celles apportées par la validité sont en bleu et celles apportées par la localité d'état sont en rouge.

cela, nous montrons que le flux fenêtré de taille k $(V + EC + SL)$-cohérent a un nombre de Consensus égal à k, ce qui veut dire qu'il peut être utilisé pour implémenter le Consensus entre k processus, mais pas plus. Comme il est impossible d'implémenter le Consensus dans les systèmes sans-attente [43], le flux fenêtré de taille k est un exemple de type de données abstrait pour lequel $(V + EC + SL)$ ne peut pas être implémenté.

Lemme 5.2. *Pour tout $k \in \mathbb{N}$, le flux fenêtré de taille k $(V + EC + SL)$-cohérent a un nombre de Consensus supérieur ou égal à k.*

Démonstration. Soit $k \in \mathbb{N}$. Nous montrons qu'il est possible d'implémenter le consensus dans $AS_k[\emptyset]$ à l'aide d'un flux fenêtré de taille k $(V + EC + SL)$-cohérent.

On considère l'algorithme de la figure 5.5 pour implémenter le Consensus : pour proposer une valeur v, un processus écrit $v + 1$ dans un flux fenêtré de taille k $(V + EC + SL)$-cohérent. Le processus p_i accède à cet objet partagé par l'accesseur local $\mathbf{flux}_i$. Écrire $v + 1$ plutôt que v permet d'assurer que toutes les valeurs écrites sont différentes de la valeur par défaut, même si la valeur proposée est nulle. Le processus retourne la première valeur non nulle de sa première lecture, à laquelle il soustrait 1.

Nous prouvons maintenant que cet algorithme implémente le Consensus. Pour cela, considérons une histoire H produite par une exécution de l'algorithme. Soit H' une histoire obtenue en ajoutant une infinité de lectures à la suite de l'écriture et de la lecture de tous les processus corrects lors de l'exécution de l'algorithme. Chaque processus p_i est dans l'une des trois situations suivantes :

- il tombe en panne avant le début de l'algorithme,
- il écrit la valeur $v_i + 1$ puis tombe en panne,
- il écrit la valeur $v_i + 1$ puis lit la suite de valeurs $\left(\zeta_j^i\right)_{j\in\mathbb{N}}$.

Cette histoire est représentée sur la figure 5.6. Si aucun processus n'est correct, le Consensus est trivialement atteint. Nous supposons donc qu'au moins un processus est correct.

Convergence. Puisque l'objet partagé est convergent, comme le nombre $|U_{T,H}| \leq k$ d'écritures est fini, il existe un état ζ_{EC} et un ensemble cofini de lectures Q_{EC} tels que toutes les lectures de Q_{EC} retournent ζ_{EC}.

Validité. Soit Q_V l'ensemble cofini de lectures de la validité. Les ensembles Q_{EC} et Q_V sont tous les deux cofinis par rapport à E_H, donc il existe une lecture $e \in Q_{EC} \cap Q_V$, faite par le processus p_i. D'après la validité, il existe une linéarisation $L_V \in \text{lin}\,(H[E_H/\{e\}]) \cap L(\mathcal{W}_k)$. Notons $f(j)$ l'identifiant du processus qui a appelé la j^{e} écriture de la linéarisation L_V. Comme L_V respecte la spécification séquentielle du flux fenêtré, e retourne la valeur $\zeta_V = (0, \cdots, 0, v_{f(1)} + 1, \cdots, v_{f(l)} + 1)$ formée de $k - |U_{T,H}|$ fois la valeur par défaut 0 suivies de $l \leq k$ valeurs écrites. Comme $e \in Q_{EC}$, on a $\zeta_{EC} = \zeta_V$.

Localité d'état. Comme le flux fenêtré vérifie la localité d'état, pour chaque processus correct p_i, il existe un ensemble d'événements C^i_{SL} qui contient toutes les lectures pures de l'histoire et une linéarisation L^i_{SL} qui contient les lectures de $C^i_{SL} \cap p_i$ et les écritures de C^i_{SL}. Soit Q^i_{SL} l'ensemble des lectures de p_i qui apparaissent après la dernière écriture dans L^i_{SL}. Encore une fois, $C^i_{SL} \cap p_i$ est cofini, $(C^i_{SL} \cap p_i) \cap Q_{EC}$ n'est pas vide, donc il existe une lecture qui retourne $\zeta_{SL} = \zeta_{EC} = \zeta_V$. Comme ζ_V contient $|U_{T,H}|$ valeurs non nulles et ζ_{SL} contient $|C^i_{SL} \cap U_{T,H}|$ valeurs non nulles, on a $C^i_{SL} \cap U_{T,H} = U_{T,H}$: la linéarisation L^i_{SL} contient toutes les écritures de l'histoire. On en déduit deux choses :

- l'écriture de p_i faisant partie de C^i_{SL}, la première lecture de p_i ne retourne pas l'état initial ;
- la taille k du flux fenêtré étant supérieure ou égale au nombre d'écritures, la première valeur non nulle est la même dans toutes les lectures de p_i.

Comme la première valeur non nulle des lectures de Q^i_{SL} est $v_{f(1)} + 1$, la première valeur non nulle de la première lecture de p_i est également $v_{f(1)} + 1$, qui ne dépend pas de i.

Revenons-en à l'algorithme. L'histoire H produite par l'algorithme est un préfixe de H'. Tout processus correct retourne $v_{f(1)}$ (terminaison), qui est la valeur commune retournée par tous les processus (accord). De plus, $v_{f(1)}$ est la valeur proposée par $p_{f(1)}$ (validité). Ainsi, l'algorithme de la figure 5.5 implémente bien le Consensus dans $AS_k[\emptyset]$ donc le flux fenêtré de taille k $(V + EC + SL)$-cohérent a un nombre de Consensus supérieur ou égal à k

□

Lemme 5.3. *Pour tout $k \in \mathbb{N}$, le flux fenêtré de taille k séquentiellement cohérent a un nombre de Consensus inférieur ou égal à k.*

Démonstration. Cette démonstration reprend la structure des preuves de l'article original sur le nombre de Consensus [56], elles-mêmes inspirées de la démonstration de l'impossibilité du Consensus dans les systèmes répartis asynchrones par passage de message où au moins une panne franche peut avoir lieu [43].

La notion centrale dans ces démonstrations est la *valence* : un état global au cours de l'exécution d'un algorithme est *v-valent* si, à partir de cet état global, l'algorithme ne peut retourner que v quel que soit l'ordonnancement des processus dû à l'asynchronisme. Un état global v-valent pour une valeur v quelconque est dit *monovalent*. Un état

global qui n'est pas monovalent est dit *plurivalent*. Un processus ne peut terminer que dans un état global monovalent. L'idée est donc de construire une exécution infinie dans laquelle l'état global reste plurivalent.

Supposons qu'il existe un algorithme qui résout le Consensus dans un système asynchrone composé de $k+1$ processus communiquant uniquement par l'intermédiaire d'un flux fenêtré de taille k séquentiellement cohérent. Initialement, tous les processus corrects appellent la méthode **propose**. Nous supposons qu'au moins deux valeurs différentes sont proposées.

Soient p_i et p_j deux processus qui proposent des valeurs v_i et v_j différentes. Dans une exécution où tous les processus sauf p_i tombent en panne avant d'avoir appelé une opération du flux fenêtré, p_i doit retourner v_i pour respecter la validité, donc v_i peut être retournée par l'algorithme. Symétriquement, v_j peut être retourné par l'algorithme, donc l'état global initial est plurivalent.

Supposons l'existence d'un état global plurivalent s tel que l'appel à la prochaine opération du flux fenêtré par n'importe quel processus résulte en un état monovalent. Deux situations sont possibles suivant si la prochaine opération d'un processus est une lecture ou non.

- Supposons qu'il existe un processus p_i dont le prochain appel soit une lecture. Sa lecture mène à un état global v_i-valent s'. Comme s est plurivalent, il existe un processus p_j dont le prochain appel mène à un état v_j-valent, avec $v_i \neq v_j$. Il existe donc une exécution depuis l'état s, dans laquelle p_i n'effectue aucun appel et p_j termine en retournant v_j. Les états globaux s et s' ne différant que par l'état local de p_i, au cours de la même exécution depuis l'état s', p_j termine également en retournant v_j, ce qui est contradictoire avec le fait que s' est v_i-valent.

- Dans l'autre cas, le prochain appel de tous les processus est une écriture. Comme l'état global est plurivalent, il existe deux valeurs différentes v_i et v_j et deux processus p_i et p_j tels que l'exécution de p_i mène à un état global v_i-valent s_i et l'exécution de p_j mène à un état global v_j valent s_j. D'un côté, l'appel successif par les $k-1$ autres processus de leur écriture à partir de l'état s_j mène dans l'état s'_j, qui est v_j-valent car s_j l'est. Il existe donc une exécution depuis l'état s'_j, dans laquelle p_i n'effectue aucun appel et p_j termine en retournant v_j. D'un autre côté, l'appel successif par p_j de son écriture puis par les $k-1$ autres processus de leur écriture à partir de l'état s_i mène dans l'état s'_i, qui est v_i-valent car s_i l'est. L'état du flux fenêtré dans s'_i et s'_j est le même, puisque les k dernières écritures sont les mêmes (celle de p_j suivie de celle des $k-1$ autres processus). Les états globaux s'_i et s'_j ne différant que par l'état local de p_i, au cours de la même exécution depuis l'état s'_i, p_j termine également en retournant v_j, ce qui est contradictoire avec le fait que s'_i est v_i-valent.

Un tel état n'existe pas, donc il est possible de construire une séquence infinie d'appels à des opérations du flux fenêtré qui mènent toutes à des états globaux plurivalents. Cela est en contradiction avec le fait que l'algorithme termine, donc un tel algorithme n'existe pas. Il est impossible d'implémenter le Consensus pour $k+1$ processus avec un flux fenêtré séquentiellement cohérent de taille k, donc le nombre de consensus du flux fenêtré de taille k séquentiellement cohérent est inférieur ou égal à k.

□

Théorème 5.4 (Remplissage de l'échelle de Herlihy)**.** *Pour tout $k \in \mathbb{N}$, les objets partagés $(V + EC + SL)(\mathcal{W}_k)$ et $SC(\mathcal{W}_k)$ ont un nombre de Consensus égal à k.*

Démonstration. Soit $k \in \mathbb{N}$. On sait que la cohérence séquentielle est plus forte que les trois critères primaires. Elle est donc plus forte que leur conjonction. On en déduit que le nombre de Consensus de $(V + EC + SL)(\mathcal{W}_k)$ est inférieur ou égal à celui de $SC(\mathcal{W}_k)$. On sait de plus d'après le lemme 5.2 que le nombre de consensus de $(V + EC + SL)(\mathcal{W}_k)$ est au moins k et d'après le lemme 5.3, que celui de $SC(\mathcal{W}_k)$ est au plus k. Finalement, les deux objets partagés ont un nombre de Consensus égal à k.

□

Corollaire 5.5 (Critère fort)**.** *La conjonction des trois critères primaires $(V + EC + SL)$ est un critère fort.*

Démonstration. Soient $n > 1$ et $k \geq 2$. On sait d'après le théorème 5.4 que le nombre de Consensus de l'objet $(V + EC + SL)(\mathcal{W}_k)$ est égal à k. On en déduit d'après [56] qu'il n'existe pas d'algorithme implémentant $(V + EC + SL)(\mathcal{W}_k)$ dans $AS_n[t \leq 1]$. Par conséquent, il n'en existe pas non plus dans $AS_n[\emptyset]$, donc $(V + EC + SL)$ est un critère de cohérence fort.

□

Le lemme 5.2 montre que les trois critères de cohérence étudiés sont liés aux trois propriétés du Consensus : la validité permet d'affirmer que la valeur retournée par l'algorithme du Consensus sera l'une des valeurs proposées (*validité* du Consensus), puisqu'elle est obtenue par une linéarisation des écritures de l'histoire ; la convergence permet l'*accord* puisque tous les processus doivent aboutir à la même valeur, et la localité d'état assure que la première valeur non nulle dans les lectures restera toujours la même, point clé utilisé pour savoir quand l'algorithme peut s'arrêter, et donc garantir la *terminaison* du Consensus. Le corollaire 5.5 permet donc de séparer l'espace des critères faibles en six grandes familles comprises entre le critère minimal $C_\perp$ et la famille des critères forts. D'un côté, la convergence, la validité et la localité d'état sont trois propriétés très faibles qui correspondent chacune à une propriété du Consensus. Ces *critères primaires* peuvent être conjugués deux à deux pour former trois nouvelles familles de *critères secondaires* qui garantissent chacune deux des trois propriétés du Consensus. De plus, chaque famille de critères secondaires possède une famille de critères primaires complémentaire, correspondant à la propriété du Consensus qu'elle ne garantit pas. La conjonction de critères de familles complémentaires est un critère fort. Cette séparation justifie la représentation sous forme de couleurs primaires et secondaires utilisée sur la figure 5.1a. Remarquons également que deux critères de familles de critères secondaires (respectivement primaires) différentes sont incomparables entre eux. Si ce n'était pas le cas, d'après la structure de treillis, leur conjonction serait le plus fort des deux, c'est-à-dire un critère faible (respectivement primaire), or il s'agit d'un critère fort (respectivement secondaire).

La figure 5.7 représente un morceau du système de transitions d'un type de données abstrait T. Les états de T sont divisés en trois sous-ensembles : T_0, T_a et T_b tels qu'il n'existe aucune transition entre des états de T_a et des états de T_b. À un moment donné, deux opérations d'écriture a et b sont effectuées et tous les processus sont dans un état de T_0 tel qu'exécuter a mène dans un état de T_a et exécuter b mène dans un état de b. La validité interdit aux processus de rester indéfiniment dans T_0 mais il est impossible de garantir dans les systèmes sans-attente que les processus iront tous dans T_a ou tous dans T_b. Dans le cas contraire, soit ils resteront séparés et l'histoire ne sera

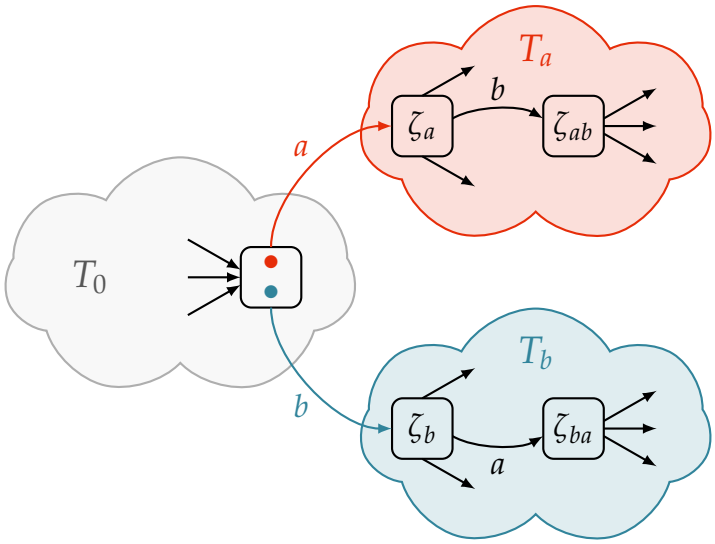


FIGURE 5.7 – Les deux écritures a et b ont été effectuées. La validité interdit aux deux jetons (qui représentent deux processus) de rester dans T_0, la localité d'état interdit à un jeton de sauter de T_a vers T_b ou inversement et la convergence impose que les deux jetons finissent dans le même état. La seule possibilité est que les deux jetons prennent la même première transition, ce qui ne peut pas être garanti.

pas convergente, soit certains devront sauter d'un état à un autre au détriment de la localité d'état. Tous les types de données abstraits qui possèdent ce motif dans leur système de transition sont sujets à ce résultat d'impossibilité. C'est en particulier le cas des services de messagerie instantanée, d'où les trois stratégies observées dans l'introduction : Hangouts reste dans T_0 en sacrifiant la validité, WhatsApp renonce à la convergence en présentant les messages dans l'ordre où il les reçoit et Skype cède sur la localité d'état en réordonnant les messages. Pour le flux fenêtré, la taille k indique le nombre minimal d'écritures nécessaires pour revenir dans un état commun depuis les états ζ_a et ζ_b : par exemple, pour un flux fenêtré de taille 2, on peut arriver dans l'état $(3,4)$ en deux écritures en partant de l'état $(0,1)$ comme de l'état $(0,2)$. L'histoire étudiée dans le lemme 5.2 contient au plus k écritures. Les premières sont utilisées pour passer de T_0 à T_a et T_b et les suivantes ne suffisent pas à revenir dans un état commun. Remarquons que plus k augmente, plus le système de transitions du flux fenêtré de taille k se rapproche de celui de la figure 5.7 : ce dernier a un nombre de Consensus infini (si au moins une lecture permet de savoir si l'on se trouve dans T_a et T_b) alors que le flux fenêtré à un nombre de Consensus de k.

5.2.2 Pluralité de la décomposition

Une question naturelle à propos de la décomposition en critères primaires et secondaires est celle de son unicité. Nous montrons qu'il n'y a pas unicité de la décomposition. Pour cela, nous construisons une famille infinie non dénombrable de critères de cohérence faibles dont la conjugaison deux à deux est un critère fort.

Le but de cette étude est de questionner les limites de notre modèle pour répondre à des questions sur l'espace des critères faibles dans son ensemble. La question de la dénombrabilité est une première illustration des questions qui se posent. La construction du théorème 5.6 ne fonctionnerait plus si l'on demandait l'existence d'un algorithme générique pour implémenter tous les types de données abstraits, car un tel algorithme

serait obligé de calculer l'écriture binaire d'un nombre réel quelconque, ce qui n'est pas possible en général. On pourrait cependant exhiber de la même manière une infinité *dénombrable* de critères faibles dont la conjugaison deux à deux est un critère fort. De plus, tel un kaléidoscope, chacun de ces critères présente des facettes des mêmes critères de cohérence (la convergence et la cohérence pipeline) vus sous un angle différent. La convergence et la cohérence pipeline semblent plus « homogènes » et donc plus « naturels » que les critères que l'on construit ici. Le principal intérêt du théorème 5.6 est donc de montrer que l'introduction de tels concepts est un prérequis pour toute approche analytique des critères faibles.

Pourtant, une telle approche est nécessaire pour répondre à des questions comme celle de l'existence de critères faibles maximaux, qui pourraient être posée de la façon suivante : la frontière séparant l'espace des critères faibles de celui des critères forts contient-elle des critères faibles ? Une façon de répondre à ces questions pourrait être de proposer une logique permettant d'exprimer les critères de cohérence.

Théorème 5.6. *Il existe une infinité non dénombrable de critères de cohérence faibles dont la conjonction deux à deux est un critère fort.*

Démonstration. Soit $n > 1$ processus. Nous construisons une famille infinie non dénombrable de critères de cohérence $(C_x)_{x\in[0;2^{-n+1}[}$ pour lesquels il existe au moins un ADT qui ne peut pas être implémenté dans $AS_n[t < 1]$.

Soit $x \in [0;2^{-n+1}[$ un nombre réel. Son écriture binaire[1] est une suite de chiffres $(x_k)_{k\geq 2}$ telle que $x = \sum_{k=2}^{\infty} x_k 2^{-k}$.

À tout x, on associe le critère de cohérence C_x qui se comporte comme la cohérence pipeline pour les flux fenêtrés dont la taille correspond à un 1 dans l'écriture de x, et à la convergence pour tous les autres types :

$$C_x : \begin{cases} \mathcal{T} & \to & \mathcal{P}(\mathcal{H}) & \\ \mathcal{W}_k & \mapsto & PC(\mathcal{W}_k) & \text{si } x_k = 1 \\ T & \mapsto & EC(T) & \text{sinon} \end{cases}$$

L'application $x \to C_x$ est injective car deux réels différents ont toujours au moins un chiffre différent dans leur écriture binaire. De plus, $[0;2^{-n+1}[$ n'est pas dénombrable, donc $\{C_x : x \in [0;2^{-n+1}[\}$ n'est pas non plus dénombrable.

Pour tout x, C_x est bien un critère faible car pour tout T, $C_x(T) \in \{PC(T), EC(T)\}$. Il ne reste donc qu'à montrer que la conjonction des C_x deux à deux est un critère fort. Soient x et y deux réels différents de $]0;2^{-n+1}[$. Il existe $k \geq n$ tel que $x_k \neq y_k$. Or $(C_x + C_y)(\mathcal{K}_k) = (PC + EC)(\mathcal{K}_k)$ qui ne possède pas d'implémentation dans $AS_n[t < 1]$ d'après le théorème 5.4. On en déduit que $C_x + C_y$ est un critère fort.

□

5.3 Hiérarchie des types de données abstraits

La partie précédente prouve qu'il n'est pas possible d'implémenter *tous* les ADT avec la conjonction des trois critères primaires. Cela ne signifie pas qu'il n'existe pas d'implémentation pour un type particulier. Par exemple, on sait qu'un registre unique séquentiellement cohérent peut être implémenté dans $AS_n[\emptyset]$, de même qu'un flux fenêtré de taille 1 qui se comporte de la même manière.

[1] Cette écriture n'est pas unique (par exemple $0,001111... = 0,010000...$) mais il nous suffit qu'il en existe une, et nous fixons une écriture pour chaque x maintenant

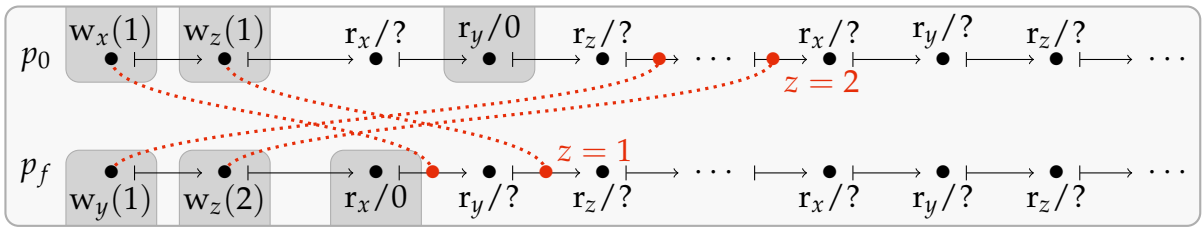


FIGURE 5.8 – Peut-on implémenter une mémoire convergente et pipeline cohérente ?

L'étude des critères de cohérence apporte un nouvel outil pour étudier la calculabilité dans les systèmes répartis. Ainsi, l'algorithmique du réparti peut-être décrite comme l'étude d'un espace à trois « dimensions » : les systèmes, les types de données abstraits et les critères de cohérence. Les principaux résultats du domaine, comme [12, 56] concernent la linéarisabilité, et les variations du critère de cohérence n'ont, à notre connaissance, jamais été étudiées dans ce but. Dans cette partie, nous nous intéressons plus spécifiquement aux critères de cohérence forts pour lesquels certains types de données abstraits (la mémoire et les types de données abstraits commutatifs) peuvent être implémentés dans les systèmes sans-attente. Nous montrons que cela permet de dégager une hiérarchie entre ces types de données abstraits, basée sur le plus fort critère qu'ils peuvent vérifier dans un système sans-attente.

5.3.1 La mémoire

Implémenter à la fois la convergence, la localité d'état et la validité pour la mémoire dans les systèmes sans-attente est en fait possible : on peut montrer que l'algorithme de la figure 3.7 (page 80) implémente la localité d'état, et donc la conjonction des trois critères faibles, puisque $SUC + SL > V + EC + SL$. Cela ne remet pas en cause la portée du résultat précédant et le découpage en familles de critères primaires et secondaires : le théorème 5.7 prouve que pour la mémoire, la cohérence pipeline (PC) et la convergence (EC) sont inconjugables.

Théorème 5.7. *Soient x, y et z trois noms de variable, $n > 1$ processus et $f \geq \frac{n}{2}$. Il n'existe pas d'algorithme implémentant $(PC + EC)\left(\mathcal{M}_{\{x,y,z\}}\right)$ dans $AS_n[t \leq f]$.*

Démonstration. Soient x, y et z trois noms de variable, $n > 1$ processus et $f \geq \frac{n}{2}$. Supposons qu'il existe un algorithme A qui implémente $(PC + EC)\left(\mathcal{M}_{\{x,y,z\}}\right)$ dans $AS_n[t \leq f]$. Nous allons montrer (preuve par l'absurde) qu'il existe une exécution admise par A qui ne respecte pas le critère $(PC + EC)$.

Les deux ensembles de processus $\Pi_0 = \{p_0, ..., p_{f-1}\}$ et $\Pi_f = \{p_f, ..., p_{n-1}\}$ forment une partition du système et chacun de ces ensembles a une taille au moins égale à $n - f$. On considère un programme dans lequel p_0 écrit 1 dans x puis 1 dans z, puis lit tour à tour x, y et z dans une boucle infinie ; p_f écrit 1 dans y puis 2 dans z, puis lit également les trois variables indéfiniment ; les autres processus n'écrivent ni ne lisent aucune variable. Tous les processus exécutent l'algorithme A. Ce programme génère l'histoire sur la figure 5.8.

Est-il possible que p_0 lise $y = 0$ et p_f lise $x = 0$ lors de leur première lecture ?

Si cela est possible, comme A implémente la cohérence pipeline, il doit exister une linéarisation de l'histoire de p_0 contenant toutes les écritures et une infinité de lectures

pour chaque variable. La première lecture de y, qui retourne 0, doit être placée avant l'écriture $\mathrm{w}_y(1)$, qui doit elle-même être placée avant l'écriture $\mathrm{w}_z(2)$ pour respecter l'ordre de programme. L'ensemble des linéarisations possibles pour p_0 est donc le langage ω-régulier L_0 défini ci-dessous. De la même manière, le langage des linéarisations possibles pour p_f est le langage L_f.

$$L_0 = \mathrm{w}_x(1) \cdot \mathrm{w}_z(1) \cdot (\mathrm{r}_x/1 \cdot \mathrm{r}_y/0 \cdot \mathrm{r}_z/1)^+ \cdot \mathrm{w}_y(1) \cdot (\mathrm{r}_x/1 \cdot \mathrm{r}_y/1 \cdot \mathrm{r}_z/1)^\star \cdot \mathrm{w}_z(2) \cdot (\mathrm{r}_x/1 \cdot \mathrm{r}_y/1 \cdot \mathrm{r}_z/2)^\omega$$

$$L_f = \mathrm{w}_y(1) \cdot \mathrm{w}_z(2) \cdot (\mathrm{r}_x/0 \cdot \mathrm{r}_y/1 \cdot \mathrm{r}_z/2)^+ \cdot \mathrm{w}_x(1) \cdot (\mathrm{r}_x/1 \cdot \mathrm{r}_y/1 \cdot \mathrm{r}_z/2)^\star \cdot \mathrm{w}_z(1) \cdot (\mathrm{r}_x/1 \cdot \mathrm{r}_y/1 \cdot \mathrm{r}_z/1)^\omega$$

Nécessairement, l'histoire contient donc une infinité d'événements étiquetés $\mathrm{r}_z/2$ et une infinité d'événements étiquetés $\mathrm{r}_z/1$. Or aucun état de $\mathcal{M}_{\{x,y,z\}}$ n'accepte ces deux lectures et l'histoire ne contient qu'un nombre fini d'écritures. La convergence ne peut donc pas être assurée. Cela signifie que A doit interdire à p_0 de lire $\mathrm{r}_y/0$ ou à p_f de lire $\mathrm{r}_x/0$.

Or cette exigence est justement celle qui ne peut pas être assurée d'après [14] et [48]. Supposons en effet les trois scénarios suivants :

S_0 : dans ce premier scénario, tous les processus de Π_0 tombent en panne dès le début de l'exécution, avant même d'envoyer un message, tandis que ceux de Π_f sont corrects. Puisque A est capable de supporter jusqu'à f fautes et que $|\Pi_0| = f$, toutes les opérations de p_f doivent terminer, et l'histoire obtenue doit être pipeline cohérente. Toutes les opérations étant séquentiellement ordonnées, la seule linéarisation possible pour p_f est décrite par $\mathrm{w}_x(1) \cdot \mathrm{w}_z(1) \cdot (\mathrm{r}_x/1 \cdot \mathrm{r}_y/0 \cdot \mathrm{r}_z/1)^\omega$.

S_f : le deuxième scénario est similaire mais les processus de Π_0 sont corrects et ceux de Π_f sont fautifs. Comme $|\Pi_f| = n - f$ et $f > \frac{n}{2}$, on a $|\Pi_f| \leq f$, donc de la même manière, la seule linéarisation possible pour p_0 est décrite par $\mathrm{w}_y(1) \cdot \mathrm{w}_z(2) \cdot (\mathrm{r}_x/0 \cdot \mathrm{r}_y/1 \cdot \mathrm{r}_z/2)^\omega$.

$S_\varnothing$: dans le dernier scénario, tous les processus sont corrects, mais les messages échangés entre les processus de Π_0 et Π_f sont différés. Du point de vue de p_0 lors de sa première lecture de y, tout ce passe exactement de la même manière que dans le cas S_f : il n'y a aucune façon de différencier un processus fautif d'un processus dont tous les messages envoyés sont très lents. p_0 doit donc se comporter comme dans S_f et retourner 0 lors de cette lecture. De même, p_f, incapable de différencier ce cas de S_0, doit retourner 0 lors de sa première lecture de x.

Finalement pour ce programme, l'algorithme A doit à la fois interdire la lecture $y = 0$ à p_0 ou la lecture $x = 0$ à p_f pour garantir $PC + EC$, et autoriser ces mêmes lectures pour tolérer les fautes. Ces deux exigences sont contradictoires, donc A n'existe pas. □

Ce résultat doit mettre en garde contre une simplification commune de la cohérence pipeline, selon laquelle ce critère, qui respecte l'ordre de processus, est un chaînon entre la cohérence causale qui respecte l'ordre causal et la convergence qui ne respecte rien. D'une part la validité et la localité d'état sont deux propriétés importantes qui ne doivent pas être ignorées. D'autre part il est faux que la cohérence pipeline et la cohérence causale renforcent la convergence, et toute tentative pour les renforcer en ajoutant la convergence à leur définition ne peut se faire qu'au prix de la perte de l'un ou l'autre des critères primaires les constituant, ou bien d'une sortie du modèle sans-attente.

5.3.2 Les types de données commutatifs

De même que le corollaire 5.5 ne signifie pas qu'aucun type de données abstrait ne peut respecter les trois critères primaires, le théorème 5.7 ne dit pas que la cohérence pipeline et la convergence ne peuvent jamais être atteintes en même temps. En particulier, dans le cas où toutes les opérations commutent, la convergence est très facile à obtenir. C'est cette intuition qui a mené au développement des CRDT [111]. Comme des objets plus complexes dont les opérations ne commutent pas (par exemple l'OR-Set) ont également été qualifiés de CRDT [112], nous introduisons le nouveau nom *CADT* (pour « Commutative Abstract Data Types », ou Types de Données Abstraits Commutatifs) pour désigner les ADT dont les opérations d'écriture sont commutatives. Parmi les CADT, on trouve par exemple le compteur non borné muni des opérations d'incrémentation et de décrémentation, ou encore l'ensemble dans lequel il n'est possible que d'insérer des éléments.

Définition 5.6 (Type de Données Abstrait Commutatif)**.** Un *CADT* est un ADT $(\mathsf{A}, \mathsf{B}, \mathsf{Z}, \zeta_0, \tau, \delta)$ tel que pour toute paire d'opérations $(\alpha, \alpha') \in \mathsf{A}^2$, et pour tout état $\zeta \in \mathsf{Z}$, $\tau(\tau(\zeta, \alpha), \alpha') = \tau(\tau(\zeta, \alpha'), \alpha)$.

Nous conjecturons que l'hypothèse d'une majorité de processus corrects est nécessaire (conjecture 5.8) et suffisante (conjecture 5.9) pour implémenter les CADT. Cela signifie que, du point de vue de l'implémentation, les CADT et la mémoire nécessitent les mêmes hypothèses. Il en résulte que, comme la mémoire, les CADT ont un nombre de Consensus égal à 1.

Conjecture 5.8. *Soient $n > 1$ processus et $f \geq \frac{n}{2}$. Il existe un type de données abstrait commutatif T tel qu'il n'existe pas d'algorithme implémentant $SC(T)$ dans $AS_n[t \leq f]$.*

Conjecture 5.9. *Soit T un type de données abstrait commutatif. Il existe un algorithme implémentant $SC(T)$ dans $AS_n[t < \frac{n}{2}]$.*

La théorie des critères de cohérence faibles met en évidence une différence de calculabilité entre la mémoire et les CADT : la conjonction de la cohérence pipeline et de la convergence peut être implémentée pour les CADT mais pas pour la mémoire.

Dans le système de transitions de la figure 5.7, un CADT vérifierait $\zeta_{ab} = \zeta_{ba}$. L'exécution de a et de b doit donc toujours mener à cet état commun, que le chemin pour y parvenir passe par ζ_a ou par ζ_b. Plutôt que de prouver que les CADT peuvent être implémentés dans les systèmes sans-attente en exhibant un algorithme, le théorème 5.10 prouve que la convergence est obtenue gratuitement pour toute histoire vérifiant la cohérence pipeline. C'est cette propriété qui justifie l'intérêt porté aux CRDT.

Remarque 5.1. L'hypothèse que l'ordre de processus est un bel ordre est importante dans la preuve du théorème 5.10. Sans cette hypothèse, l'existence d'une antichaîne infinie de lectures faites dans l'état initial, différent de l'état de convergence, serait techniquement possible, ce qui empêcherait d'assurer la convergence.

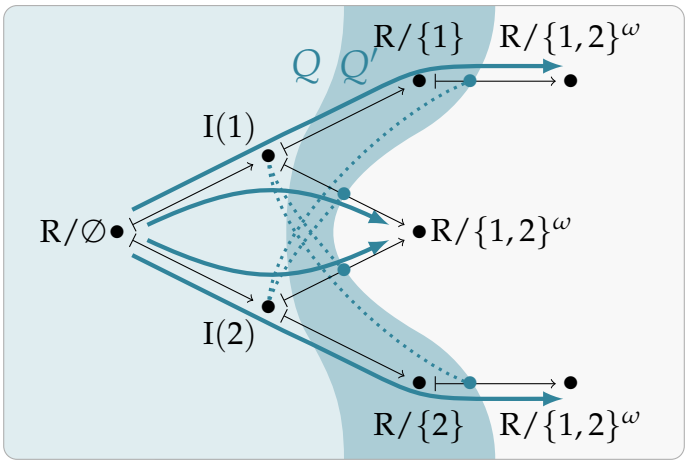


FIGURE 5.9 – Pour les CADT, la cohérence pipeline implique la convergence.

Théorème 5.10 (Implémentation des CADT). *Pour tout CADT T, $PC(T) \subset EC(T)$.*

Démonstration. Soient T un CADT et $H \in PC(T)$. Si H possède une infinité d'écritures, alors $H \in EC(T)$. Supposons que H possède un nombre fini d'écritures, comme sur la figure 5.9.

Comme H vérifie la cohérence pipeline, pour toute chaîne maximale $p \in \mathscr{P}_H$, il existe $l_p \in \text{lin}(H[E_H/p]) \cap L(T)$. Comme le nombre d'écritures est fini, il existe un préfixe fini de l_p qui les contient toutes. Soit Q_p l'ensemble des lectures de l_p contenues dans ce préfixe et $Q = \bigcup_{p \in \mathscr{P}_H} Q_p$.

Montrons maintenant que Q est fini. Toutes les chaînes contenues dans Q sont contenues dans un Q_p particulier et sont donc finies. Q peut donc être mis sous la forme $Q = \cup_{e \in Q'} \lfloor e \rfloor_{\mapsto}$, où Q' est l'ensemble des éléments maximaux de Q. Comme $\mapsto$ est un bel ordre, Q', qui est une antichaîne de $(E_H, \mapsto)$, est fini et pour tout $e \in Q'$, $\lfloor e \rfloor_{\mapsto}$ est fini. Q, en tant qu'union finie d'ensembles finis, est fini.

Comme T est un CADT, tous les chemins étiquetés par les écritures de H mènent au même état ζ. Tout événement de lecture pure $e \in E_H \setminus Q$ apparaissant dans une linéarisation l_p pour un p donné est bien fait dans l'état ζ, donc $H \in EC(T)$.

□

5.4 Quel critère utiliser ?

On sait maintenant qu'il n'existe pas de plus fort critère de cohérence faible et que les trois critères secondaires ne sont pas comparables. Mais peut-être que l'un d'entre eux, sans être plus fort que les autres, offre des garanties plus intéressantes pour un programme que les autres ? En fait, chaque critère secondaire a ses propres avantages, mais aussi ses propres faiblesses. C'est pourquoi il est important de bien choisir le critère le mieux adapté à l'application.

Sérialisabilité. Dans l'exemple des services de messagerie instantanée, la sérialisabilité correspond à Hangouts (page 11), où le message de Bob qui n'a pas été transmis peut être qualifié d'avorté. Le principal problème de la sérialisabilité est qu'elle ne garantit pas du tout le progrès : l'implémentation qui fait avorter

toutes les opérations d'écriture et retourne l'état initial pour toutes les opérations de lecture pure est sérialisable. Cela en fait un critère relativement faible. En contrepartie, les événements avortés sont connus par le processus qui les a initiés. Cela signifie que la sérialisabilité laisse à l'utilisateur le soin de gérer les fautes, en ne faisant que lui signaler les opérations qui n'ont pas pu aboutir.

Dans l'exemple des services de messagerie instantanée, on comprend que la sérialisablité est le critère le plus utilisé sur Internet. Les services implémentés sur le modèle client-serveur garantissent naturellement l'ordre total sur toutes les opérations (l'ordre total du serveur). En contrepartie, les opérations appelées hors connexion sont perdues, mais le processus appelant est averti de cette perte. La vérification de la connexion est généralement effectuée en utilisant une minuterie[2]. Le coût de la sérialisabilité sera donc assez élevé en pratique, à la fois en temps d'exécution (une attente de réponse du serveur est nécessaire) et en tolérance aux fautes (le serveur est un point individuel de défaillance). L'existence d'algorithmes plus efficaces sans serveur reste un problème ouvert.

Finalement, la sérialisabilité est surtout à conseiller quand l'application demande un très grand contrôle des données, par exemple pour effectuer une transaction bancaire, et que le système offre une stabilité suffisante pour garantir une proportion d'événements avortés assez faible, ce qui correspond bien au modèle client-serveur, où le serveur est supposé suffisamment fiable.

Cohérence d'écritures. À la fois Skype (page 13) et Facebook Messenger (page 70) montrent un comportement admis par la cohérence d'écritures. Dans ces deux exemples, des messages ont été réordonnés, soit à la réception d'un nouveau message (Skype), soit ultérieurement de façon asynchrone (Facebook Messenger). Dans ce cas, l'absence de la localité d'état n'a pas beaucoup d'importance car tous les messages restent visibles. En général, la cohérence d'écritures peut autoriser des lectures successives de paraître complètement incohérentes les unes par rapport aux autres. En particulier, au sein d'un algorithme réparti, les valeurs écrites dépendent généralement des valeurs lues, ce qui rend la cohérence d'écritures difficile à utiliser dans ces cas là.

L'impossibilité d'ajouter la localité d'état à la cohérence d'écritures a de plus un impact sur le coût de cette dernière. En effet, aucun algorithme implémentant la cohérence d'écritures ne peut garantir la localité d'état. Or comme son nom l'indique, ce dernier critère est l'abstraction du fait que chaque processus maintient un état local qui évolue en fonction des écritures conformément à la spécification séquentielle. Ce résultat condamne donc les implémentations de la cohérence d'écritures à utiliser des techniques plus compliquées pour maintenir l'état sur lequel sont faites les lectures, d'où la liste des messages reçus stockée en mémoire et les messages de correction envoyés dans les algorithmes du chapitre 3.

Finalement, les critères de la famille de la cohérence d'écritures sont surtout adaptés au plus haut niveau, pour les applications réparties elles-mêmes, comme les éditeurs collaboratifs. Dans ce cas, l'utilisateur de ces objets partagés est un être humain plus apte à s'adapter à de petites incohérences qu'un algorithme.

[2]Une attente de minuterie ne contredit pas notre définition de « sans-attente » à base de tolérance aux pannes franches, mais on aperçoit ici que de subtils changements dans une définition peut mener à des systèmes différents.

	PC, *UC* ou *Ser*	$EC + SL + V$	$EC + PC$	*SC*
$\mathcal{W}_{k,(k\geq n)}$	$AS_n[\emptyset]$	$AS_n[t = 0]$	$AS_n[t = 0]$	$AS_n[t = 0]$
$\mathcal{M}_{\{x,y,z\}}$	$AS_n[\emptyset]$	$AS_n[\emptyset]$	$AS_n\left[t < \frac{n}{2}\right]$	$AS_n\left[t < \frac{n}{2}\right]$
$\mathcal{M}_{\{x,y\}}$	$AS_n[\emptyset]$	$AS_n[\emptyset]$	?	$AS_n\left[t < \frac{n}{2}\right]$
CADT	$AS_n[\emptyset]$	$AS_n[\emptyset]$	$AS_n[\emptyset]$	$AS_n\left[t < \frac{n}{2}\right]$
$\mathcal{M}_x, \mathcal{W}_1$	$AS_n[\emptyset]$	$AS_n[\emptyset]$	$AS_n[\emptyset]$	$AS_n[\emptyset]$

FIGURE 5.10 – Système minimal nécessaire pour implémenter différents objets. Les objets qui peuvent être implémentés sans-attente sont représentés en bleu.

Cohérence Pipeline. La cohérence pipeline correspond à la stratégie exhibée dans l'expérience sur WhatsApp (page 12). Le problème ici était qu'Alice et Bob ne voyaient pas les messages dans le même ordre à la fin, ce qui correspond à l'absence de convergence. Cette absence est rédhibitoire pour de nombreuses applications, en particulier celles gérant des données géo-répliquée, dans lesquelles le maintient d'un état cohérent entre les différents centres de réplication est un enjeu majeur.

Les critères de cette famille ont en revanche un coût d'implémentation bien plus faible que ceux des deux autres familles, puisqu'il suffit de diffuser un message à chaque écriture et que l'état local à chaque processus est seulement un état abstrait (figure 4.14).

Finalement, la cohérence pipeline est surtout adaptée dans les cas où les processus utilisent les valeurs qu'ils lisent pour décider des prochaines valeurs à écrire, par exemple dans les applications surtout centrées sur le calcul, ainsi que pour implémenter des algorithmes répartis comme l'entrée dans une section critique, dont la nécessaire terminaison rend la convergence (à plus long terme) moins importante.

Conclusion

Dans ce chapitre, nous avons étudié la structure de l'espace des critères de cohérence faibles. Un critère de cohérence faible est un critère de cohérence pour lequel n'importe quel type de données abstrait peut être implémenté dans le système $AS_n[\emptyset]$, formé de n processus asynchrones communiquant par passage de messages et sujet à un nombre arbitraire de pannes franches.

Nous avons montré qu'il n'existait pas de plus grand élément parmi les critères faibles. L'espace des critères de cohérence faibles peut être représenté comme l'ensemble des couleurs visibles (figure 5.1a). Comme les couleurs primaires et secondaires, trois familles de critères secondaires ont chacune une famille de critères pri-

maires complémentaire. La conjonction d'un critère primaire et de son critère secondaire complémentaire est un critère fort. Même s'il n'y a pas unicité sur cette décomposition, celle proposée nous paraît particulièrement intéressante. Premièrement, chaque critère primaire correspond à une propriété du Consensus. Deuxièmement, les trois stratégies observées dans l'expérience de l'introduction sur les services de messagerie instantanée reflètent chacune une famille de critères secondaires. Enfin, tous les critères faibles étudiés dans cette thèse se rangent naturellement dans l'une de ces familles.

Nous nous sommes également posés la question des types de données abstraits pour lesquels il était possible d'implémenter un critère fort. La figure 5.10 dresse le bilan des résultats obtenus. En faisant varier le critère de cohérence, on peut mettre en évidence une hiérarchie dans les types de données abstraits. À système constant, plus le critère de cohérence demandé est fort, moins d'objets peuvent être implémentés en vérifiant ce critère. Pour le flux fenêtré de taille au moins 2, la conjonction des trois critères faibles est impossible à implémenter dans un système à passage de messages asynchrone où au moins une panne franche peut se produire. Implémenter la conjonction de la convergence et de la cohérence pipeline pour une mémoire composée d'au moins trois registres nécessite l'hypothèse d'une majorité de processus correcte dans chaque exécution. Il est en revanche possible d'implémenter un unique registre séquentiellement cohérent dans les systèmes sans-attente. Savoir si la conjonction de la convergence et de la cohérence pipeline peut être implémentée pour une mémoire composée de deux registres reste une question ouverte. Enfin, nous avons prouvé que la conjonction de la convergence et de la cohérence pipeline pouvait être implémentée pour les types de données abstraits commutatifs dans les systèmes sans-attente. Il serait assez facile de prouver que la cohérence séquentielle ne peut pas être atteinte pour ces objets dans un tel système. Tous ces résultats montrent que la théorie des critères faibles offre un nouvel angle d'étude pour comprendre la calculabilité dans les systèmes répartis.

6

La bibliothèque CODS

Sommaire

Introduction

L'une des raisons d'être de la science est d'identifier les abstractions les plus adaptées pour résoudre un problème donné. Dans le domaine de l'informatique, l'introduction de langages de programmation de plus en plus hauts niveaux a permis de simplifier grandement la programmation et, ce qui va de paire, d'étendre de plus en plus la gamme d'applications effectivement produites. L'invention des langages dédiés tels HTML, PHP et JavaScript a permis l'apparition puis le développement du Web. Pour les systèmes répartis décentralisés, les efforts ont surtout porté sur la cohérence forte incarnée par des abstractions telles les exclusions mutuelles, le Consensus, voire même la mémoire transactionnelle. Les abstractions manquent toujours pour la cohérence faible, d'où les efforts considérables à fournir, encore aujourd'hui, pour développer la moindre application dans les systèmes qui ne peuvent garantir plus, comme les systèmes pair-à-pair ou l'informatique en nuage. La question naturelle qui se pose est donc la suivante.

Problème. *Quelle forme pourrait prendre un langage dédié aux systèmes sans-attente ?*

L'approche suivie dans cette thèse nous semble intéressante pour remplir ce rôle car elle replace l'aspect fonctionnel, grâce à la spécification séquentielle, au cœur de la spécification des objets répartis. La principale force de cette approche est qu'elle se base sur les notions bien établies de *systèmes de transitions*, *automates* et *langages* pour décrire les spécifications séquentielles. Ces concepts mathématiques sont centraux dans

de nombreux champs de l'informatique et de nombreux outils ont été développés pour leur spécification, modélisation et vérification. Replacer la spécification séquentielle au cœur de la spécification des objets partagés revient à affirmer que tous ces outils peuvent être adaptés dans les systèmes répartis, au moins pour décrire et vérifier l'aspect fonctionnel des programmes. Par exemple, tous les critères basés sur l'existence d'une linéarisation correcte vis-à-vis de la spécification séquentielle (dont tous ceux renforçant la validité ou la localité d'état) garantissent que tous les états lus sont accessibles dans le système de transitions. Si ce dernier assure un invariant, le même invariant sera automatiquement respecté par l'objet partagé. La programmation orientée objet est sans doute le plus utilisé des outils basés sur les systèmes de transitions. La spécification d'*objets* séquentiels, sous forme de *classes* ou de *structures*, est naturelle dans ce paradigme.

Approche. *Nous proposons de réutiliser sans modification tous les aspects de la programmation orientée objets relative à la spécification des spécifications séquentielles, et de ne gérer la concurrence que par le choix d'un critère de concurrence.*

Pour illustrer la pertinence de cette approche, nous avons développé une bibliothèque de critères de cohérence, CODS (pour « Concurrent Objects in Distributed Systems »). Un critère de cohérence dans CODS peut être vu comme un foncteur qui transforme une classe décrivant une spécification séquentielle, en une autre classe permettant l'accès à un objet partagé vérifiant ce critère de cohérence. Ainsi, un programmeur peut se consacrer à la description de l'architecture fonctionnelle des objets de son programme sans se soucier de l'aspect réparti, puis laisser CODS gérer la concurrence. De plus, CODS offre un cadre permettant de définir de nouveaux critères de cohérence. Les avantages de l'approche sont multiples.

1. L'utilisation de CODS simplifie grandement le développement d'applications réparties : un objet séquentiel est beaucoup plus simple à programmer qu'un objet partagé. En effet, le programmeur de la spécification séquentielle peut se concentrer sur l'aspect fonctionnel uniquement, et s'abstraire totalement des problèmes liés à la concurrence.

2. La lisibilité et l'évolutivité du code sont largement améliorées. Par exemple, si un même type de données abstrait peut être utilisé dans plusieurs contextes (par exemple, un ensemble partagé pour contenir des informations utiles à tous les processus, et un ensemble non partagé utilisé dans des calculs locaux au sein d'une fonction), la même classe peut être utilisée dans tous les contextes, seul le critère de cohérence change.

3. L'approche promet, à terme, une plus grande fiabilité : la programmation répartie étant beaucoup plus sujette aux bogues que la programmation séquentielle, il peut s'avérer un choix raisonnable de confier la gestion de la concurrence à une bibliothèque externe potentiellement mieux vérifiée que ne le serait un programme gérant lui-même la concurrence.

4. Enfin, on pourrait espérer à terme gagner en efficacité d'exécution. Il en va comme avec les compilateurs : un code compilé est aujourd'hui souvent beaucoup mieux optimisé qu'un code en assembleur écrit à la main. On pourrait espérer de même que les applications suivant cette approche puissent profiter plus facilement des progrès accomplis sur les algorithmes génériques pour implémenter les critères de cohérence.

La bibliothèque présentée ici est un prototype développé pour le langage D [33], un langage de la famille de C dont la syntaxe est très proche de celle de Java et de C#. Le choix de ce langage a été réalisé dans la perspective d'offrir l'interface la plus transparente possible pour décharger au maximum le programmeur de la gestion de l'aspect réparti. Le langage D possède les trois caractéristiques suivantes, nécessaires au développement d'un tel projet.

1. D est un langage de programmation orienté objets, ce qui est un prérequis pour adapter un travail sur les objets partagés de manière cohérente.
2. Le langage D offre des techniques de réflexivité assez poussées, permettant l'introspection des classes comparables à celle disponible en Java. Cette fonctionnalité est nécessaire pour étudier les méthodes de la classe de la spécification séquentielle.
3. La métaprogrammation est également un point fort de D. Cette fonctionnalité est nécessaire pour créer, à la compilation, les classes des objets partagés et dont les méthodes sont celles de la classe de la spécification séquentielle, inconnue *a priori*.

Contribution. *La contribution présentée dans ce chapitre est la bibliothèque CODS. Elle peut être découpée en la résolution de deux défis principaux. D'une part, la capture des appels aux méthodes des objets à partager, qui permet l'intégration élégante des critères de cohérence dans un langage orienté objets tel que D, et d'autre part la conception d'un cadre de programmation permettant de définir de nouveaux critères. Ce travail a été réalisé en collaboration avec Olivier Ruas [104]. Le code de la bibliothèque est disponible en ligne [83].*

Ce chapitre se présente comme un tutoriel de prise en main de la bibliothèque. La partie 6.1 présente l'utilisation de la bibliothèque du point de vue du programmeur d'une application répartie qui veut utiliser des critères de cohérence. Ensuite, la partie 6.3 montre comment la bibliothèque peut être étendue en définissant de nouveaux critères de cohérence.

6.1 Vue d'ensemble

La bibliothèque CODS a été conçue pour rendre son utilisation aussi transparente que possible. En fait, en dehors de la configuration du réseau, qui n'est pas prise en charge par la bibliothèque, la différence entre une application séquentielle simple et son homologue répartie basée sur CODS tient essentiellement en une ligne : l'appel du constructeur dans le programme séquentiel est remplacé par une connexion au critère de cohérence dans le programme réparti.

L'implémentation d'objets partagés *via* CODS suit le même schéma que leur spécification formelle : elle est séparée en une implémentation séquentielle (une classe D) et un choix de critère de cohérence (parmi ceux proposés dans la bibliothèque). L'appel du critère de cohérence paramétré par la spécification séquentielle crée un objet partagé qui présente exactement la même interface que l'objet séquentiel.

L'utilisation basique de la bibliothèque est illustrée par un exemple sur la figure 6.1. La figure 6.1a montre un programme en D qui utilise CODS. Après inclusion des bibliothèques externes, le code du programme peut être séparé en trois sections : la classe `WindowStream` qui encode une spécification séquentielle très proche de $\mathcal{W}_2$, la fonction `p` qui décrit l'exécution des processus qui utilisent un objet partagé, et la fonction principale `main` qui lance le programme.

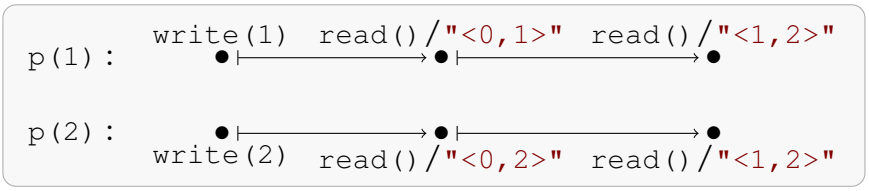
```d
/******************** Bibliothèques externes ********************/
import std.stdio, std.conv, core.thread; // Bibliothèque standard D
import cods;                             // Outils généraux de CODS
import networkSimulator;                 // Simulateur du réseau
import uc;                               // Cohérence d'écritures

/******************* Spécification séquentielle *******************/
class WindowStream {
  private int x = 0, y = 0;              // État
  public void write(int val) {           // Écriture pure
    x=y; y=val;
  }
  public string read() {                 // Lecture pure
    return "<"~to!string(x)~","~to!string(y)~">";[a]
  }                                      // Retourne <x,y>
}

/****************** Code exécuté par les processus ******************/
void p(int i) { // i=1 ou i=2
  // Initialisation de l'instance partagée
  WindowStream ws = UC.connect!WindowStream("ws1");
  // Utilisation de l'instance partagée
  ws.write(i);                           // Écriture
  writeln(ws.read());                    // Première lecture
  Thread.sleep(dur!("msecs")(1000));     // Attente de la convergence
  writeln(ws.read());                    // Deuxième lecture
}

/*************** Lancement des processus parallèles ***************/
void main () {
  // Déclaration des processus
  auto network = new NetworkSimulator!2([{ p(1); }, { p(2); }]);
  // Configuration de CODS
  Network.configure(network);
  // Démarrage des processus
  network.start();
}
```

(a) Programme réparti utilisant un objet partagé $\mathcal{W}_2$ en D.

[a] en D, $\sim$ désigne la concaténation de chaînes de caractères et ! désigne l'appel d'un template.

```
<0,1>
<0,2>
<1,2>
<1,2>
```

(b) Sortie standard.



(c) Analyse de l'exécution sous forme d'histoire concurrente.

FIGURE 6.1 – Exemple d'utilisation de la bibliothèque CODS.

Initialisation du réseau. Pour fonctionner, CODS a besoin d'un réseau sous-jacent capable de diffuser des messages de manière fiable. La bibliothèque en elle-même ne gère pas cet aspect de la programmation concurrente. Tout ce qu'elle fournit est une interface `INetwork` qui requiert l'implémentation de deux fonctions.

- **void** `broadcast(immutable(void)[] file)` assure la diffusion fiable du message `file` à tous les processus du réseau,
- **int** `getID()` fournit un identifiant unique au processus appelant.

La ligne 34 doit être présente dans tout programme utilisant CODS. Il s'agit de l'initialisation du réseau utilisé par le programme. Dans cet exemple, nous utilisons la classe `NetworkSimulator` qui simule un réseau asynchrone à partir de processus UNIX. Les lignes 32 et 36 sont relatives à cette classe. La seule information importante les concernant est qu'elles lancent deux processus en parallèle. L'un exécute la fonction `p(1)` et l'autre exécute la fonction `p(2)`.

Spécification séquentielle. Ces deux processus utilisent un objet de type `WindowStream` dont la classe est définie entre les lignes 8 et 16. Une telle classe encode un ADT. L'ensemble des états est l'espace de définition des variables membres, que nous abstrayons ici en $\mathbb{N}^2$. L'état initial est la valeur par défaut des variables membres, dans ce cas $(0, 0)$. L'alphabet d'entrée A contient les méthodes membres avec toutes les combinaisons possibles pour leurs arguments. On retrouve donc dans A les symboles `read()` et `write(n)` pour tout $n \in \mathbb{N}$. L'alphabet de sortie B est l'union des valeurs de retour possibles pour toutes les méthodes, le type **void** étant remplacé par un type unitaire contenant une seule valeur $\bot$. Dans cet exemple, B contient $\bot$ et toutes les chaînes de caractères de la forme `"<x,y>"`, où $x, y \in \mathbb{N}$. Enfin, la fonction de transition et la fonction de sortie sont définies par l'effet de bord et la valeur de retour des méthodes de la classe. L'ADT spécifié par la classe `WindowStream` est donc très proche de $\mathcal{W}_2$ (voir page 29), au renommage des opérations et au type de sortie de la lecture près.

Critère de cohérence. La classe `WindowStream` est utilisée par la fonction `p(i)` exécutée par les deux processus. L'application de la bibliothèque est visible à la ligne 21. Si celle-ci était remplacée par `WindowStream ws = new WindowStream();` cette fonction n'aurait rien de particulier. Elle commencerait par créer un objet de type `WindowStream`, écrirait son argument dedans puis lirait deux fois à une seconde d'intervalle. La seule particularité apportée par CODS ici est la façon d'initialiser les objets partagés. L'instruction `UC.connect!WindowStream("ws1")` contient trois informations.

- `UC` est le critère de cohérence utilisé. Ici, il s'agit de la cohérence d'écritures. Pour pouvoir utiliser ce critère, il est nécessaire d'importer la bibliothèque correspondante, ce qui est fait à la ligne 5. Pour l'instant, seules la cohérence d'écritures et la cohérence pipeline ont été implémentées.
- `WindowStream` est la classe à paralléliser.
- `"ws1"` est l'identifiant unique de l'objet partagé sur le réseau. Dans cet exemple, les deux processus ont initialisé leur instance avec le même identifiant, donc leurs opérations concernent le même objet partagé. Si un processus se connecte plusieurs fois au même identifiant unique, il obtiendra le même objet.

```
1  void p (int i) {
2    WindowStream ws1 = UC.connect!WindowStream("ws1");
3    WindowStream ws2 = UC.connect!WindowStream("ws2");
4    if(i==1) {                              // code pour p(1)
5      ws1.write(2);
6      ws2.write(3);
7    } else {                                // code pour p(2)
8      ws2.write(4);
9      ws1.write(5);
10   }
11   Thread.sleep(dur!("msecs")(500));   // attente de la convergence
12   // convergence dans l'état ws1 = <5,2> et ws2 = <3,4> interdite
13   writeln(ws1.read() ~ " " ~ ws2.read());
14 }
```

FIGURE 6.2 – Programme utilisant un seul couple composé de deux flux fenêtrés.

Une façon d'interpréter cette instruction est d'imaginer que tous les objets pré-existent virtuellement dans le réseau. L'instruction ne crée qu'un accesseur permettant de les manipuler. Cet accesseur est de type `UC.Type!WindowStream`, qui hérite de `WindowStream`. Cela permet une intégration parfaite de l'objet dans son environnement, puisqu'aucune autre modification dans le reste du programme n'est nécessaire.

Le programme n'est pas déterministe. La sortie affichée sur la figure 6.1b est l'une des sorties possibles. Ce qui s'est passé pendant cette exécution peut être expliqué par une histoire concurrente (figure 6.1c). Le processus qui exécute `p(1)` a écrit 1 dans l'objet partagé puis lu `"<0,1>"` avant convergence et `"<1,2>"` après convergence (l'attente d'une seconde a été suffisante pour atteindre la convergence dans tous les cas testés). Celui qui exécute `p(2)` a écrit 2 puis lu `"<0,2>"` et `"<1,2>"`. Il y a donc bien eu échange d'information entre les processus, et l'histoire obtenue vérifie la cohérence d'écritures mais pas la cohérence séquentielle. La cohérence d'écritures est garantie pour toutes les exécutions utilisant le critère de cohérence `UC`.

6.2 Les transactions

Comme on l'a vu dans le chapitre 2, la composabilité est une propriété rare des critères de cohérence. Il serait trop restrictif de n'accepter que les critères composables. Pour contourner l'absence de composabilité, CODS compose les spécifications séquentielles avant l'application du critère de cohérence. Ainsi, au lieu d'appliquer le critère de cohérence directement sur chacune des spécifications séquentielles et d'espérer qu'ils se comportent mutuellement comme si le critère s'appliquait à la composition des spécifications, CODS compose les spécifications séquentielles avant d'appliquer le critère de cohérence. En d'autres termes, si un programme utilise deux ADT T_1 et T_2 avec un critère de cohérence C, CODS implémentera $C(T_1 \times T_2)$ plutôt que $C(T_1, T_2)$. On notera ITT la composition des spécifications séquentielles de tous les objets utilisés dans un programme.

Lorsque la fonction `p` du programme de la figure 6.1 est remplacée par celle de la figure 6.2, les deux processus gèrent deux objets partagés, ws_1 et ws_2 ayant pour identifiants respectifs `"ws1"` et `"ws2"`. Le processus qui exécute $p(1)$ écrit 2 dans ws_1 puis 3

dans ws_2, et l'autre processus écrit 4 dans ws_2 puis 5 dans ws_1. Après convergence, il est impossible d'avoir à la fois ws_1 dans l'état $(5, 2)$ et ws_2 dans l'état $(3, 4)$, car l'histoire obtenue serait dans $UC(ws_1, ws_2)$ mais pas dans $UC(ws_1 \times ws_2)$.

Transactions. La composition des spécifications séquentielles implémentée dans CODS permet de gérer correctement la composition sans nuire à la modularité, car elle est transparente. Elle autorise également une fonctionnalité avancée de CODS : les *transactions*. Une transaction dans CODS est un morceau du programme qui permet d'encapsuler une succession d'opérations sur les objets partagés dans un événement unique de l'histoire concurrente. Du point de vue du critère de cohérence, une transaction se comporte exactement comme une opération. La seule différence est qu'elle n'est pas définie directement dans la classe des objets partagée, mais généralement avec le code qui régit le comportement des processus. Dans CODS, l'ensemble Γ des transactions définies dans le programme s'applique à la composition ΠT des spécifications séquentielles des objets partagés, ce qui crée un nouvel ADT $\Pi T[\Gamma]$, dans lequel toutes les transaction ont été transformées en opérations. La formalisation du concept de transaction et de leur interaction avec les objets partagés est exposée dans la définition 6.1.

Définition 6.1. Soit un ADT $T = (\mathsf{A}, \mathsf{B}, \mathsf{Z}, \zeta_0, \tau, \delta) \in \mathcal{T}$. Une *transaction* sur T est un triplet $\gamma = (\mathsf{B}_\gamma, \tau_\gamma, \delta_\gamma)$ où :

- B_γ est un ensemble dénombrable appelé *alphabet de sortie* ;
- $\tau_\gamma : \mathsf{Z} \to \mathsf{Z}$ est la fonction de transitions ;
- $\delta_\gamma : \mathsf{Z} \to \mathsf{B}_\gamma$ est la fonction de sortie.

Soit Γ un ensemble dénombrable de transactions sur T. L'application de Γ sur T est l'ADT

$$T[\Gamma] = (\mathsf{A}' = \mathsf{A} \cup \Gamma, \mathsf{B}' = \mathsf{B} \cup \bigcup_{\gamma \in \Gamma} \mathsf{B}_\gamma, \mathsf{Z}, \zeta_0, \tau', \delta')$$

où :

$$\tau' : \begin{cases} \mathsf{Z} \times \mathsf{A}' & \to & \mathsf{Z} & \\ (q, \alpha) & \mapsto & \tau(q, \alpha) & \text{si } \alpha \in \mathsf{A} \\ (q, \gamma) & \mapsto & \tau_\gamma(q) & \text{si } \gamma \in \Gamma \end{cases} \qquad \delta' : \begin{cases} \mathsf{Z} \times \mathsf{A}' & \to & \mathsf{B}' & \\ (q, \alpha) & \mapsto & \delta(q, \alpha) & \text{si } \alpha \in \mathsf{A}. \\ (q, \gamma) & \mapsto & \delta_\gamma(q) & \text{si } \gamma \in \Gamma \end{cases}$$

CODS propose deux types de transactions : les *transactions nommées*, les plus générales et les *transactions anonymes*, à la fois plus simples d'utilisation et plus restreintes.

Transactions nommées. Les transactions nommées sont un mécanisme qui permet de définir toutes les transactions faisant figurer des objets partagés. Une transaction nommée est une classe qui hérite de la classe abstraite `Transaction!T`, où `T` est le type de retour de la transaction (B_γ). Elle implémente la méthode `T execute()` qui définit le comportement de la transaction.

Le programme de la figure 6.3 illustre l'utilisation des transactions nommées. Outre les inclusions de bibliothèques, la définition de la spécification séquentielle `Registre(T)` et la fonction principale du programme, similaires au programme de la figure 6.1, le programme contient la définition et l'appel de la transaction `Increment`.

```
/******************** Bibliothèques externes ********************/
import std.stdio, std.conv, core.thread;
import cods;
import networkSimulator;
import uc;

/******************* Spécification séquentielle *******************/
class Register(T) {
  private T state = 0;
  public T write(T val) {          // Écriture
    state=val;
    return val;
  }
  public T read() {                // Lecture pure
    return state;
  }
}

/************** Spécification de la transaction nommée *************/
class Increment : Transaction!void { // void : type de retour
  private int v;
  public this(int val) {           // Constructeur
    v = val;
  }
  public override void execute() {  // Obligatoire
    Register!int x = UC.connect!(Register!int)("RegX");
    x.write(v + x.read());         // int tmp = x.read(); x.write(v+tmp)
        ;
  }
}

/***************** Code exécuté par les processus *****************/
void p(int i) {
  Register!int x = UC.connect!(Register!int)("RegX");
  // Utilisation de la transaction
  UC.transaction!void(new Increment(i));
  Thread.sleep(dur!("msecs")(500));
  writeln(x.read());
}

/*************** Lancement des processus parallèles ***************/
void main () {
  Network.registerType!Increment; // Déclaration de la transaction
  auto network = new NetworkSimulator!2([{p(1);}, {p(2);}]);
  Network.configure(network);
  network.start();
}
```

FIGURE 6.3 – Implémentation de l'incrémentation à l'aide d'une transaction nommée.

```
1 void p(int i) {
2   WindowStream x = UC.connect!WindowStream("x");
3   UC.anonymousTransaction({            // Début de la transaction
4     x.write(i);
5     x.write(i);
6   });                                  // Fin de la transaction
7   Thread.sleep(dur!("msecs")(500));
8   writeln(x.read());                   // "<1,1>" ou "<2,2>" possibles
9 }
```

FIGURE 6.4 – Exemple de transaction anonyme : dans l'état final, les deux champs ont la même valeur.

- La transaction nommée `Increment` est définie entre les lignes 20 et 29. Cette transaction permet d'incrémenter un registre d'une valeur `val` passée en argument du constructeur. Le corps de la fonction `execute()` contient l'écriture et la lecture d'un registre, nécessaires à son incrémentation. La connexion à ce registre est faite de manière habituelle, à la ligne 26.

- À la ligne 35, la fonction `C.transaction!T(Transaction!T t)` est utilisée pour appeler la transaction. `C` est le critère de cohérence et `T` est le type de retour de la transaction `t` (**void** dans le cas de la transaction `Increment`). Remarquons qu'il est nécessaire de déclarer le type de la transaction au démarrage du programme. Cela est fait à la ligne 42 dans l'exemple.

Dans ce programme, un processus incrémente le registre de 1 et l'autre de 2. Si l'appel de la transaction était remplacé par l'instruction `x.write(i + x.read())`, la lecture et l'écriture des deux processus pourraient être entrelacés de sorte que le registre converge vers 1, 2 ou 3. En encapsulant le calcul dans une transaction, on garantit que la valeur de convergence sera forcément 3.

Transactions anonymes. Définir une nouvelle classe pour chaque transaction peut parfois complexifier excessivement le code source d'un programme. Les *transactions anonymes* sont un mécanisme léger permettant la définition des transactions définissant des *écritures pures* sous la forme d'une *succession d'opérations indépendantes*.

Une transaction anonyme peut être définie à l'endroit même où elle est appelée. Un exemple est présenté sur la figure 6.4. Pour créer une transaction anonyme, il est nécessaire d'appeler la fonction `C.anonymousTransaction(void delegate()dg)` où `C` est le critère de cohérence désiré. En D, le type `RT delegate(T)` désigne une fonction anonyme dont le type de retour est `RT` et dont l'argument est de type `T`. L'objet `dg` est une fonction anonyme dont le corps, défini entre accolades, contient la suite des appels d'opérations, qui compose la transaction. Dans le programme de la figure 6.4, il ne peut y avoir d'entrelacement entre les écritures des deux processus. L'état final ne peut donc être que $(1,1)$ ou $(2,2)$ alors que sans la transaction, on aurait pu converger vers $(1,2)$ ou $(2,1)$.

Les transactions anonymes sont plus simples à utiliser que les transactions nommées, mais leur utilisation est également beaucoup plus restreinte. Cela est dû à une restriction de la bibliothèque de sérialisation pour le langage D, Orange [30] que l'on utilise pour créer les messages, et qui ne s'applique pas aux objets de type `delegate`.

Une transaction anonyme est obligatoirement formée d'une succession d'opérations indépendantes sur des objets partagées. En particulier, la valeur retournée par une opération ne peut pas être utilisée comme argument d'une opération suivante. Par exemple, la transaction de la figure 6.3 ne pourrait pas être écrite comme une fonction anonyme, car la valeur écrite dépend de la valeur lue. Les transactions anonymes ont tout de même un intérêt, principalement pour préparer un état en empêchant certaines situations de compétition. Par exemple, si un programme gère deux ensembles partagés A et B et veut toujours garantir que $A \subset B$, il doit toujours insérer les valeurs dans B quand il veut les insérer dans A, et les supprimer dans A pour les supprimer dans B. Il peut utiliser des transactions anonymes pour éviter les entrelacements du type $A.\mathrm{D}(x) \cdot A.\mathrm{I}(x) \cdot B.\mathrm{I}(x) \cdot B.\mathrm{D}(x)$, qui mènent à un état interdit. Pour la même raison, une transaction ne peut pas retourner de valeurs, puisque l'instruction **return** n'est pas une opération sur un objet partagé. C'est pourquoi les transactions anonymes ne peuvent être que des écritures pures.

6.3 Définition de nouveaux critères

CODS définit également un cadre permettant d'implémenter de nouveaux critères de cohérence. La figure 6.5 montre l'implémentation de la cohérence pipeline dans ce cadre. L'utilisation de ce critère ne diffère de celle de la cohérence d'écritures que dans l'import des bibliothèques externes (**import** `pc;` remplace **import** `uc;`) et dans le critère utilisé lors des appels aux fonctionnalités de CODS (`PC.connect`, `PC.anonymousTransaction` et `PC.transaction`).

La classe `PC`, justement, est définie ligne 48 comme un alias (plus précisément un héritage vide) à `ConsistencyCriterionBase!PC_Implementation`. La classe `ConsistencyCriterionBase` fournit l'interface de CODS. Elle se charge de capturer les appels aux méthodes des objets partagés, d'encapsuler ces appels et les transactions sous une forme commune, et de les transmettre à l'implémentation du critère de cohérence. Elle gère également la concurrence locale au processus pour garantir l'exécution atomique des fonctions de l'implémentation du critère. Le travail principal à fournir pour implémenter le critère est la classe `PC_Implementation`, définie entre les lignes 12 et 46.

Celle-ci contient l'algorithme qui implémente le critère de cohérence. Elle doit implémenter la classe imbriquée `SharedObject!T` et la méthode `executeOperation()` pour étendre la classe abstraite `ConsistencyCriterionImplementation`. Quand une méthode ou une transaction est exécutée au niveau du programme utilisateur, la méthode `executeOperation` est appelée localement avec un paramètre qui encode l'opération complexe. L'opération peut être exécutée (localement ou non) grâce à sa méthode `execute()` (lignes 23 et 37). Cette exécution génère la succession des méthodes à appeler sur les objets. C'est alors la méthode `SharedObject!T.executeMethod()` qui est appelée sur l'objet correspondant. Par exemple, à chaque exécution de la transaction définie dans le programme de la figure 6.3, la méthode `executeOperation` est appelée une fois par le processus qui a ordonné la transaction et la méthode `SharedObject!Register.executeMethod()` est appelée deux fois sur chaque processus : une fois pour la lecture et une fois pour l'écriture. Dans le cas de la cohérence pipeline, la classe `SharedObject!T` est très simple. Elle encode l'état local d'un objet partagé par un état abstrait de son ADT (ligne 42) et se contente d'appliquer les méthodes sur cet état local quand elle les reçoit (ligne 45). Remarquons que le type de retour, `ExtObject`, permet de contenir tous les types, y compris **void** et les types de base.

```
1  import cods;
2  /*************** Message à diffuser sur le réseau ***************/
3  class PC_Message : Message {
4    private int id, cl; private Operation op; // Données
5    this(int mId, int mCl, Operation mOp) { // Constructeur
6      id=mId; cl=mCl; op=mOp;
7    }
8    override void on_receive() {            // Appelé à la réception
9      PC.getInstance().getImplementation().receiveMessage(id, cl, op);
10 }}
11 /************ Classe principale : implémentation de PC ************/
12 class PC_Implementation : ConsistencyCriterionImplementation {
13   private int[int] clock;                  // Horloge vectorielle
14   private Operation[int][int] pending;     // Tampon de messages
15   this() {                                 // Constructeur
16     Network.registerType!PC_Message;       // Enregistrement du type
17     clock = [Network.getInstance().getID() : 0];
18   }
19   // Une opération ou transaction a été appelée localement
20   ExtObject executeOperation(Operation op) {
21     int id = Network.getInstance().getID();
22     Message m = new PC_Message(id, clock[id]+1, op);
23     ExtObject o = op.execute();            // Exécution locale pour
24     clock[id] = clock[id]+1;               // l'atomicité de la réception
25     Network.getInstance().broadcast(m); // Diffusion du message
26     return o;
27   }
28   // Un message PC_Message(mId, mCl, mOp) a été reçu
29   void receiveMessage(int mId, int mCl, Operation mOp) {
30     if(!(mId in clock)) clock[mId] = 0;  // Premier message de mId
31     if (clock[mId] < mCl) {              // Si pas déjà  reçu
32       pending[mId][mCl] = mOp;           // Ajout du message au tampon
33       // tant qu'il reste des messages dans le tampon
34       for(int cl = clock[mId] + 1; cl in pending[mId]; cl++){
35         clock[mId] = cl;                 // Livraison du message
36         pending[mId].remove(cl);         // Nettoyage du buffer
37         pending[mId][cl].execute();      // Exécution locale
38   }}}
39   // Représentation interne des objets partagés
40   override static public class SharedObject(T) :
41       ConsistencyCriterionImplementation.SharedObject!T {
42     T t = new T();                       // État local d'une instance
43     // Fonction Appelée à chaque méthode d'une transaction
44     override public ExtObject executeMethod(Functor!T f) {
45       return f.execute(t);               // Exécution locale
46 }}}
47 /************* Définition de PC comme une abréviation *************/
48 class PC : ConsistencyCriterionBase!PC_Implementation {};
```

FIGURE 6.5 – Implémentation de la convergence pipeline : fichier `pc.d`

La classe `PC_Implementation` est principalement une implémentation de la diffusion FIFO à partir de la diffusion fiable. Quand une opération est effectuée, un message `PC_Message(mId, mCl, mOp)` est diffusé juste après son exécution locale. Pour pouvoir être envoyé, un message doit étendre la classe abstraite `Message` et fournir une méthode `on_receive()`. À la réception du message, la méthode `receiveMessage` est appelée. L'implémentation de la réception FIFO est standard. Elle se base sur un vecteur `clock` qui indique le nombre de messages reçus de chaque processus et un tampon `pending` qui stocke les messages qui ont été reçus mais dont les écritures correspondantes n'ont pas encore été appliquées. Notons que la classe du message doit également être enregistrée (ligne 16) pour que son décodage soit possible.

Conformément à la politique de composition des spécifications séquentielles de CODS, tous les objets sont implémentés conjointement. Cette stratégie peut tout de même être contraignante pour les critères qui ne sont pas décomposables. Dans ce cas, il est possible que certains objets pris séparément ne vérifient pas le critère de cohérence. C'est au développeur du critère de cohérence de faire attention que l'implémentation choisie empêche ce cas de se produire. Dans le cas de la cohérence pipeline, cela ne pose pas de problème puisque le critère est décomposable. Par contre, la cohérence d'écritures ne l'est pas, et l'algorithme $UC[k]$ ne garantit pas que chaque objet d'une composition vérifiera la cohérence d'écritures. La recherche d'un algorithme efficace et décomposable pour implémenter la cohérence d'écritures est encore une question ouverte.

Conclusion

Dans ce chapitre, nous avons présenté la bibliothèque CODS, qui montre comment la théorie sur les critères de cohérence faibles peut s'inscrire dans un langage de programmation tel que D. La conception de la bibliothèque a suivi deux buts. D'une part, son interface est la plus simple possible : dans les programmes les plus simples, seule l'instanciation des objets est affectée par la bibliothèque. D'autre part, la bibliothèque est faite pour être extensible : un cadre permet d'implémenter de nouveaux critères et la configuration du réseau reste à la charge du développeur.

Hélas cette adaptativité risque d'être la première limite de l'approche. L'algorithme le plus efficace pour implémenter un critère dépend du système sur lequel le programme s'exécute. Par exemple, dans un système parallèle synchrone, on peut savoir à tout instant qu'on ne recevra pas de messages plus vieux qu'une certaine date, ce qui simplifie grandement l'implémentation de la cohérence d'écritures. À l'inverse, dans un système à la fois plus complexe et plus souple comme un réseau pair-à-pair, on pourrait chercher à adapter le réseau pour obtenir plus d'efficacité.

L'autre limite est qu'il est actuellement impossible d'adapter l'algorithme en fonction des objets à implémenter. Or on sait que certains objets comme l'ensemble et la mémoire possèdent des structures particulières qui permettent d'importantes optimisations. Les lectures et écritures pures devraient également faire l'objet d'un traitement différent. Une solution serait d'ajouter des balises comme `@update` et `@query` pour déclarer les propriétés des opérations dans la spécification séquentielle. Pour aller vraiment plus loin et avoir autant de latitude d'action qu'un compilateur, l'intégration des critères de cohérence au sein même du compilateur ou la conception d'un langage dédié semble nécessaire. Cela permettrait également de revenir sur des désagréments moins liés à des choix de conception qu'à des difficultés techniques imposées par le langage D : l'obligation d'enregistrer les types des transactions et des messages et les restrictions imposées sur les transactions anonymes.

7

Une sémantique concurrente pour Orc

Sommaire

Introduction

Dans les définitions des chapitres précédents, nous avons toujours considéré les systèmes d'exécution des programmes comme des modèles abstraits, avec lesquels la seule interaction possible était l'*observation* des histoires concurrentes qu'ils produisaient. Les systèmes répartis formés d'un nombre fixe et connu de processus communiquant par messages a servi d'illustration à la plupart des critères de cohérence et de leurs propriétés. Dans ce modèle très simple, seules les notions de *séquentialité* et de *concurrence* sont présentes, au détriment de celles de *conflit* et de *préemption* pourtant présentes dans des systèmes plus complexes tels que l'orchestration de services Web. Dans ce chapitre, nous allons voir que le modèle introduit dans le chapitre 2 est également adapté pour décrire ces systèmes plus complexes.

Plusieurs langages ont été proposés pour programmer des applications basées sur l'orchestration de services Web. BPEL [10] est certainement le plus connu d'entre eux.

Ce travail est basé sur Orc [66, 67], un langage d'orchestration construit autour de la notion de *sites*, une généralisation des fonctions qui peut encapsuler n'importe quelle sorte de service Web, ainsi qu'un calcul formel, le *calcul Orc*, formé de quatre opérateurs qui encodent les différents comportements d'un système réparti.

La façon standard de décrire la sémantique d'un langage informatique comme Orc est la *sémantique opérationnelle structurelle* (SOS) [96]. Les spécifications en SOS prennent la forme d'un ensemble de règles d'inférence qui définissent les transitions valides à partir d'un élément lexical complexe en fonction de celles valides à partir de ses composantes. Une suite de transitions valides représente un comportement possible pour un programme. Plusieurs comportements sont possibles en cas d'existence de règles non-déterministes. Dans le cas de Orc, une telle sémantique produit un ensemble de traces séquentielles, dont chacune représente un comportement possible [65].

Cette vision est limitée pour un langage comprenant du parallélisme. En fait, dans ces langages, les concepts de concurrence, de causalité et de conflit sont importants à expliciter, par exemple pour répondre aux questions qui nécessitent une analyse des causes racines, pour pouvoir rejouer des exécutions pour aider au débogage, pour détecter les situations de compétition ou pour prouver des propriétés sur la qualité de service. Les histoires concurrentes définies dans le chapitre 2 sont des candidates naturelles pour remplacer les séquences d'événements produites par la sémantique SOS. C'est le rôle des sémantiques concurrentes d'identifier l'information supplémentaire nécessaire à la construction de ces objets.

Dans un contexte concurrent et non-déterministe, cette analyse ne peut pas être basée seulement sur la structure statique du programme et doit être faite pendant l'exécution. Ces dépendances sont aussi très difficiles à extraire sans information supplémentaire des histoires séquentielles, dans lesquelles le parallélisme est encodé, comme le non-déterminisme, par un entrelacement des événements. En effet, si l'on observe que les deux événements e_1 et e_2 peuvent se produire lors de l'exécution d'un programme dont la sémantique contient les deux histoires séquentielles $e_1 \cdot e_2$ et $e_2 \cdot e_1$, il est impossible de savoir si le programme choisit aléatoirement s'il doit exécuter e_1 puis e_2 ou l'inverse (non-déterminisme), ou bien si les deux événements sont exécutés en parallèle (concurrence).

> **Problème.** *Comment définir formellement la sémantique d'un langage de programmation concurrent pour capturer les dépendances entre les événements sous forme d'histoires concurrentes qui représentent les exécutions pendant qu'elles se déroulent ?*

Ce problème est particulièrement important pour l'analyse de propriétés sur la qualité de service ou de propriétés non fonctionnelles dépendant de chemins critiques dans les relations de dépendances [100].

Nous considérons un programme Orc déjà analysé syntaxiquement et réécrit sous sa forme intermédiaire dans le calcul Orc. Un événement dans l'exécution de ce programme correspond à l'application d'une règle dans la sémantique standard. Les appels de sites et les publications sont des exemples d'événements. L'ordre entre les événements est imposé par trois sortes de dépendances :

- la séquentialité imposée par le flux de contrôle du programme défini par la sémantique des opérateurs de Orc et capturée par l'ordre de processus ;
- la préemption imposée par l'opérateur *prune* ($<x<$) de Orc, capturée par l'ordre de programme ;

- les dépendances incluant les appels de sites conservant des données comme les lectures et écritures dans un registre. Ces dépendances externes ne font pas partie de la description de Orc, mais dépendent du critère de cohérence utilisé pour décrire ces sites. Dans ce chapitre, nous ne cherchons pas à décrire directement ces dépendances. Les sites eux-mêmes pouvant ensuite être spécifiés comme un objet partagé en utilisant les outils présentés dans les chapitres précédents.

Approche. *Nous proposons d'étendre la sémantique SOS standard en réécrivant des expressions étendues dans lesquelles de l'information supplémentaire a été ajoutée pour encoder le passé des sous-éléments selon l'ordre de processus et l'ordre de programme. Cette information est rendue visible en étendant l'étiquetage des transitions.*

Capturer la séquentialité (l'ordre de processus) et la préemption (l'ordre de programme) est difficile. En effet, ces relations sont globales et donc difficiles à localiser dans l'écriture syntaxique d'une expression à un pas donné. La solution est de conserver cette information dans un contexte associé à chaque élément syntaxique. Les règles sont réécrites de façon à lier les différents contextes pendant leur exécution. Le but est qu'une telle sémantique instrumentée reproduise le comportement standard du programme tout en calculant les informations supplémentaires nécessaires pour extraire les relations entre événements produits lors de l'exécution.

Contribution. *Ce chapitre présente la définition d'une nouvelle sémantique pour le langage Orc. Il s'agit d'une instrumentation de la sémantique standard qui préserve le comportement de Orc en capturant les relations de séquentialité et de préemption entre les événements.*

Ce résultat a été présenté à MSR 2013 [89] sous la forme d'un poster puis à NETYS 2015 [91] sous la forme d'un article.

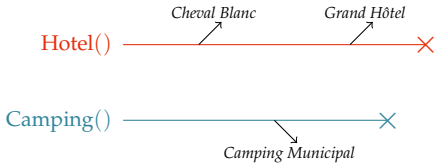
La suite de ce chapitre est organisée comme ceci. La partie 7.1 présente le langage Orc du point de vue de son calcul formel et sa sémantique opérationnelle, et l'illustre en utilisant un exemple d'orchestration de services Web. La section 7.2 détaille notre contribution : la sémantique instrumentée. La section 7.3 revient sur l'exemple de la section 7.1 pour montrer l'histoire concurrente obtenue par l'exécution de la sémantique instrumentée et comment elle peut être utilisée pour détecter les erreurs. Avant de conclure, la partie 7.4 présente les travaux connexes.

7.1 Le langage Orc

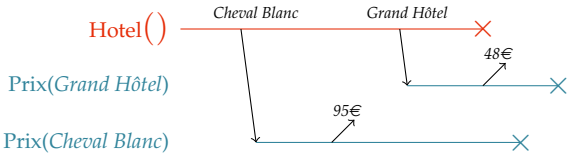
7.1.1 Calcul à grande échelle

Le langage Orc fut initialement conçu pour orchestrer les sites Web. Sa philosophie sous-jacente est que la modularité est essentielle dans la construction d'applications modernes à grande échelle. De telles applications sont un assemblage de plusieurs composants, éventuellement écrits dans différents langages, qui ne peuvent pas travailler ensemble sans un chef d'orchestre qui conduit leur exécution.

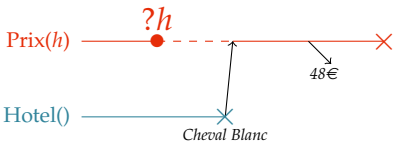
Dans Orc, ces composants, appelés *sites externes*, peuvent être n'importe quelle sorte de programmes externes, comme des services Web, ou des encapsulations d'expressions Orc. Les sites ressemblent à des fonctions dans les langages fonctionnels : ils peuvent être appelés en leur fournissant des arguments. À la différence des fonctions, ils ne retournent pas un résultat mais *publient* zéro, une ou plusieurs valeurs. Par



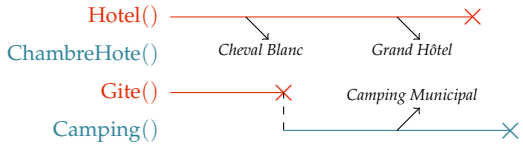
(a) Hotel() | Camping()



(b) Hotel() >h> Prix(h)



(c) Prix(h) <h< Hotel()



(d) (Hotel() ; ChambreHote()) | (Gite() ; Camping())

FIGURE 7.1 – Exemples d'exécution d'expressions Orc.

exemple, un site d'agence de voyage peut publier une offre différente pour chaque destination. Les sites peuvent être d'ordre supérieur, c'est-à-dire que les valeurs publiées peuvent elles-mêmes être des sites. Une bibliothèque standard est définie pour traiter les principaux aspects de la programmation quotidienne, comme des types complexes et des horloges qui permettent de synchroniser des parties du programme.

Les interactions entre les sites sont au cœur du langage Orc. Pour cela, un ensemble de structures de contrôle est offerte aux développeurs. Si certaines, comme les structures conditionnelles et les boucles, sont très classiques dans leur syntaxe comme dans leur comportement, celles nécessaires à la description de la concurrence sont plus originales. Ces dernières sont au nombre de quatre. Elles se basent sur les publications pour exprimer le flot de données entre les sites externes. Des exemples d'expressions Orc utilisant ces quatre opérateurs sont montrés sur la figure 7.1.

La composition *parallèle* exprime la concurrence pure. Dans $f \mid g$, f et g sont exécutés en parallèle, leurs événements sont entrelacés et l'expression s'arrête quand f et g ont tous les deux terminé. Les publications de $f \mid g$ sont l'union des publications de f et de celles de g. Ainsi, comme le montre la figure 7.1a, l'expression Hotel() | Camping() publie aussi bien les hôtels publiés par le site externe Hotel() que les campings publiés par le site externe Camping().

Comme son nom le suggère, l'opérateur *séquentiel* exprime la séquentialité. Dans l'expression $f >x> g$, la variable x peut être utilisée dans g. Initialement, f est exécuté seul. À chaque publication de v par f, une nouvelle instance de g est lancée, dans laquelle x est liée à v. Par exemple, sur la figure 7.1b, le site Hotel() publie deux valeurs, donc dans l'expression Hotel() $>h>$ Prix(h), le site Prix est appelé deux fois.

L'opérateur *prune* permet d'exprimer la préemption. Dans $f <x< g$, la variable x peut cette fois être utilisée dans f. Au début de l'exécution, f et g sont démarrés tous les deux en même temps. Quand la valeur de x est vraiment nécessaire pour avancer, f est mis en pause jusqu'à ce que g publie une valeur qui est alors liée à x dans f. Cette publication provoque l'arrêt de g : les autres événements qui auraient pu être produits par g sont préemptés par la publication. Par exemple, dans l'expression $x <x< (1 \mid 2)$, $1 \mid 2$ est sensé publier les deux valeurs 1 et 2. Or la première publication préempte la seconde, donc seule l'une des deux valeurs sera finalement publiée à chaque exécution. Ces deux publications sont en conflit. Sur la figure 7.1c, bien que le site Hotel() peut pu-

blier deux valeurs, la deuxième est préemptée donc l'expression Prix(h) $<h<$ Hotel() ne publie le prix que d'un seul hôtel.

Enfin, le dernier opérateur est appelé *otherwise*. Dans f ; g, f est démarré seul, puis g est lancé si et seulement si f s'arrête sans publier de valeur. Une utilité est de désigner la séquentialité entre des expressions qui ne publient pas de valeurs. Une autre utilisation standard est la définition de valeurs par défaut utilisées en cas d'absence de réponse d'un site. La figure 7.1d illustre les deux comportements possibles. Dans la sous-expression Hotel() ; ChambreHote(), le site Hotel() publie deux valeurs, donc ChambreHote() n'est jamais appelé. Au contraire, dans la sous-expression Gite() ; Camping(), le site Gite() termine sans publier de valeur. Le site Camping() est donc appelé.

Au plus haut niveau, une expression Orc se comporte également comme un site en publiant des valeurs. Le langage permet de définir des *sites internes* qui encapsulent des expressions Orc grâce à la syntaxe **def** $y(x_1, \cdots, x_n) = f$ où :

- y est le nom du site interne ;
- $x_1, \cdots, x_n$ sont les arguments du site qui sont évalués de manière paresseuse ;
- f est l'expression Orc à exécuter quand le site est appelé.

Les définitions de sites sont récursives, ce qui permet la même expressivité que n'importe quel langage fonctionnel.

7.1.2 Le calcul Orc

Toutes les constructions de haut niveau du langage sont uniquement définies à partir du calcul Orc, un sous-ensemble du langage Orc constitué uniquement des sites (internes et externes) et des quatre opérateurs exprimant la concurrence. Nous considérons uniquement le calcul dans le reste de ce chapitre.

Les expressions de ce calcul, regroupées dans l'ensemble Orc_s, sont définies récursivement par la grammaire de la figure 7.2a. Sont autorisées les simples publications (p), les appels ($p(p)$) et définitions ($D\#f$) de sites ainsi que l'utilisation des quatre opérateurs présentés dans la partie 7.1.1. Les expressions du calcul qui correspondent à de vrais programmes Orc sont celles qui ne contiennent ni $?k$ ni $\bot$. Ces deux symboles supplémentaires ne peuvent apparaître que pendant l'exécution d'un programme Orc.

Les règles de la sémantique standard sont définies sur la figure 7.2b. Un pas peut être effectué d'une expression Orc f vers une autre g s'il y a une règle dont f est de la même forme que le membre gauche de la conclusion, g correspond au membre droit d'une règle et que les prémisses de cette règle sont vérifiées. Le pas est étiqueté par l'étiquette de la conclusion. On distingue plusieurs types d'étiquette.

- Les étiquettes $!v$ indiquent la publication de la valeur v.
- Les étiquettes ω indiquent la terminaison du programme.
- Les étiquettes cachées, notées n de manière générique, indiquent l'exécution d'un calcul interne. On distingue à nouveau plusieurs types.
  - Les étiquettes $h(!v)$ indiquent qu'une sous-expression a publié la valeur v, mais cette publication a servi à activer un opérateur. Par exemple, dans le programme $1 >x> x + 2$, la publication du 1 par le membre de gauche n'est pas une publication de l'expression complète : une publication cachée $h(!1)$ aura donc lieu avant l'appel du site $+$ puis la publication $!2$.

f, g, h	$\in$	Expression	$::=$	$p \| p(p) \| ?k \| f \mid g \| f >x> g \| f <x< g \| f \,;\, g \| D\#f \| \bot$
D	$\in$	Définition	$::=$	$\textbf{def}\ y(x) = f$
v	$\in$	Valeur	$::=$	$V \| D$
p	$\in$	Paramètre	$::=$	$v \| \textbf{stop} \| x$
w	$\in$	Réponse	$::=$	$NT(v) \| T(v) \| Neg$
n	$\in$	Étiquette cachée	$::=$	$?V_k(v) \| ?D \| h(\omega) \| h(!v)$
l	$\in$	Étiquette	$::=$	$!v \| n \| \omega$

(a) Syntaxe des expressions du calcul Orc.

$$\text{(PUBLISH)}\ \frac{}{v \xrightarrow{!v} \textbf{stop}}\ v \text{ liée}$$

$$\text{(STOP)}\ \frac{}{\textbf{stop} \xrightarrow{\omega} \bot} \qquad \frac{}{\textbf{stop}(p) \xrightarrow{\omega} \bot}\ \text{(STOPCALL)}$$

$$\text{(EXTSTOP)}\ \frac{}{V(\textbf{stop}) \xrightarrow{\omega} \bot} \qquad k \text{ fraîche}\ \frac{}{V(v) \xrightarrow{?V_k(v)} ?k}\ \text{(EXTCALL)}$$

$$\text{(DEFDECLARE)}\ \frac{[D/y]f \xrightarrow{l} f'}{D\#f \xrightarrow{l} f'}\ D \text{ est } \textbf{def}\ y(x) = g$$

$$\text{(INTCALL)}\ \frac{}{D(p) \xrightarrow{?D} [D/y][p/x]g}\ D \text{ est } \textbf{def}\ y(x) = g$$

$$\text{(PARLEFT)}\ \frac{f \xrightarrow{l} f'}{f \mid g \xrightarrow{l} f' \mid g}\ l \neq \omega \qquad \frac{?k \text{ receives } T(v)}{?k \xrightarrow{!v} \textbf{stop}}\ \text{(REST)}$$

$$\text{(PARRIGHT)}\ \frac{g \xrightarrow{l} g'}{f \mid g \xrightarrow{l} f \mid g'}\ l \neq \omega \qquad \frac{?k \text{ receives } NT(v)}{?k \xrightarrow{!v} ?k}\ \text{(RESNT)}$$

$$\text{(PARSTOP)}\ \frac{f \xrightarrow{\omega} \bot \quad g \xrightarrow{\omega} \bot}{f \mid g \xrightarrow{\omega} \bot} \qquad \frac{?k \text{ receives } Neg}{?k \xrightarrow{\omega} \bot}\ \text{(RESNEG)}$$

$$\text{(OTHV)}\ \frac{f \xrightarrow{!v} f'}{f \,;\, g \xrightarrow{!v} f'} \qquad \frac{f \xrightarrow{!v} f'}{f >x> g \xrightarrow{h(!v)} (f' >x> g) \mid [v/x]g}\ \text{(SEQV)}$$

$$\text{(OTHN)}\ \frac{f \xrightarrow{n} f'}{f \,;\, g \xrightarrow{n} f' \,;\, g} \qquad \frac{f \xrightarrow{n} f'}{f >x> g \xrightarrow{n} f' >x> g}\ \text{(SEQN)}$$

$$\text{(OTHSTOP)}\ \frac{f \xrightarrow{\omega} \bot}{f \,;\, g \xrightarrow{h(\omega)} g} \qquad \frac{f \xrightarrow{\omega} \bot}{f >x> g \xrightarrow{\omega} \bot}\ \text{(SEQSTOP)}$$

$$\text{(PRUNEV)}\ \frac{g \xrightarrow{!v} g'}{f <x< g \xrightarrow{h(!v)} [v/x]f} \qquad l \neq \omega\ \frac{f \xrightarrow{l} f'}{f <x< g \xrightarrow{l} f' <x< g}\ \text{(PRUNELEFT)}$$

$$\text{(PRUNEN)}\ \frac{g \xrightarrow{n} g'}{f <x< g \xrightarrow{n} f <x< g'} \qquad \frac{g \xrightarrow{\omega} \bot}{f <x< g \xrightarrow{h(\omega)} [\textbf{stop}/x]f}\ \text{(PRUNESTOP)}$$

(b) Règles de la sémantique opérationnelle.

FIGURE 7.2 – La sémantique opérationnelle.

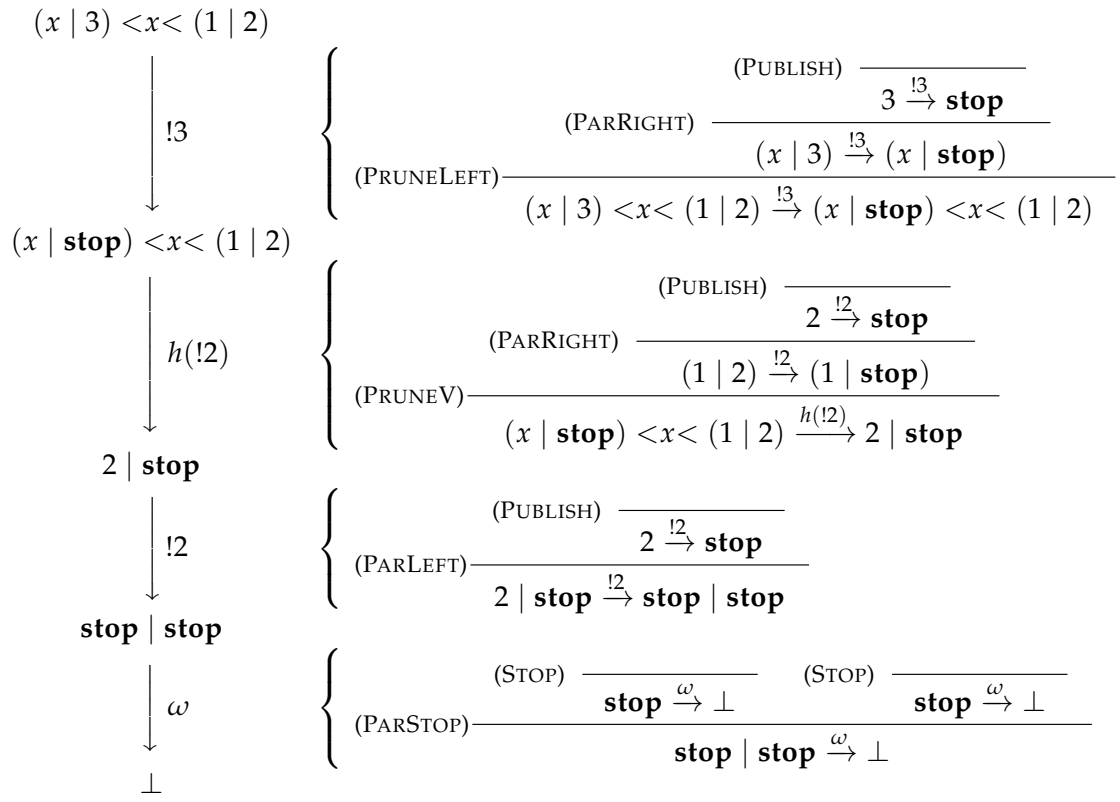


FIGURE 7.3 – Exemple d'exécution d'un programme Orc par la sémantique standard.

- Les étiquettes $h(\omega)$ indiquent la terminaison d'une sous-expression mais qui, de la même manière que les publications, peuvent être rattrapées par un opérateur (voir par exemple la règle PRUNESTOP)
- Les étiquettes $?V_k(v)$ et $?D$ indiquent l'appel d'un site externe ou interne, respectivement.

Une exécution pour f est un chemin qui part de f et effectue une succession finie ou infinie de pas valides. Une histoire séquentielle est le mot formé par les étiquettes de l'exécution, et la sémantique $[\![f]\!]$ de f est l'ensemble des histoires séquentielles admises par f. Par exemple, la figure 7.3 montre un exemple d'exécution pour le programme $(x \mid 3) <x< (1 \mid 2)$. On peut déduire de cette exécution le fait que l'histoire séquentielle $!3 \cdot h(!2) \cdot !2 \cdot \omega \in [\![(x \mid 3) <x< (1 \mid 2)]\!]$.

Les seules valeurs considérées dans le calcul Orc sont les sites, externes (V) ou internes (D). Par simplicité, nous considérons ici que les sites sont curryfiés, donc un site ne prend qu'un seul argument. Les deux sortes de sites peuvent être appelés avec leur argument en utilisant la notation fonctionnelle classique. Cependant, elles ont un comportement différent, comme on peut le voir dans la sémantique opérationnelle, ou le comportement des sites internes (INTCALL) et externes (EXTCALL, EXTSTOP, NTRES, TRES, NEGRES) est défini par des règles différentes. La principale différence est que l'appel des sites externes est strict, c'est-à-dire que leurs arguments doivent être liés avant l'appel, alors que les sites internes peuvent être appelés immédiatement et leurs arguments sont ensuite calculés de façon paresseuse. Quand un site externe est appelé, il envoie ses réponses à une marque substitutive $?k$. Une réponse peut être soit une valeur non-terminale $NT(v)$ si d'autres réponses sont attendues, soit une valeur terminale $T(v)$ si v est la dernière publication du site, soit Neg si le site termine sans publier de valeur. Le symbole **stop** peut être utilisé par le programmeur exactement

comme un site ou une variable pour marquer un programme déjà terminé. À l'exécution, **stop** produit tout de même un événement ω pour signifier à ses parents qu'il a terminé avant de se transformer en $\bot$, l'expression finale inerte.

Toutes les autres fonctionnalités du langage peuvent être exprimées dans ce calcul. Les constantes sont des sites particuliers qui se publient eux-mêmes, quels que soient leurs arguments.

Les structures de contrôle aussi sont définies à partir des sites. Ainsi, la structure `if (C) then P else Q end if` est du sucre syntaxique pour le programme Orc `(((ift(b)>> P)| (iff(b)>> Q))<b< C)`, dans lequel le site `ift` (respectivement `iff`) publie une valeur **signal** si, et seulement si, son paramètre est vrai (respectivement faux). Quant aux boucles, elles peuvent être implémentées par des sites internes récursifs.

Les structures de données complexes peuvent également être encodées à l'aide de sites uniquement. Prenons l'exemple du registre. La notation pointée `r.write(v)`, qui appelle l'écriture de la valeur `v` dans le registre `r`, n'existe pas dans la syntaxe de la figure 7.2a. Il s'agit en fait d'une abréviation de `(w(x)<w< r("write"))`. En fait, l'instance `r` peut publier ses accesseurs quand il est appelé avec leur nom. Ainsi, `r("write")` publie un site lié à `w` grâce à l'opérateur prune. Le site `w` ne publie jamais rien mais permet d'écrire une nouvelle valeur dans `r`. De la même manière, cette valeur est lisible par le site publié par `r("read")`. Quant au registre `r` proprement dit, il est publié par le constructeur `Registre` de la structure de données. Par simplicité, nous ne rentrerons pas dans ce niveau de détail par la suite, et considérerons `r.write(x)` comme un simple appel de site.

7.1.3 Illustration

Nous illustrons maintenant le langage grâce à l'exemple de la figure 7.4a. Ce programme définit le site interne `find_best(agencies, destination)` qui calcule les meilleurs offres proposées par les agences de voyages de la liste `agencies` pour la destination du paramètre `destination`. Il publie comme unique valeur une paire composée de la meilleur réponse étoffée d'informations complémentaires, et de la liste des autres offres triées par prix.

Le calcul est effectué à l'aide des trois sites internes auxiliaires `find_offers()`, `extend_best()` et `sort_offers()` qui communiquent par l'intermédiaire de trois objets partagés, créés aux lignes 21 à 23 : la file `offers` et les registres `best_offer` et `best_agency`. La ligne 22 illustre l'initialisation typique d'un objet partagé en Orc. L'instance est premièrement créée lors de l'appel du constructeur `Register()`. Elle est ensuite liée à la variable `r`, initialisée à une valeur par défaut **null** puis publiée pour être liée à la variable `best_offer` dans le reste du programme. Comme tout ceci est effectué à droite d'un opérateur *prune*, toutes les utilisations de ces objets dans les sites auxiliaires ont bien lieu après leur initialisation, mais les calculs ne dépendant pas des objets partagés peuvent être débutés en parallèle.

Le corps du site `find_best(agencies, destination)` proprement dit est composé des lignes 18 et 19. La racine de l'expression Orc est l'opérateur $>t>$ qui divise le calcul en deux étapes. Dans la première, `find_offers` et `timer(2000)` s'exécutent en parallèle jusqu'à ce que l'une des deux expressions publie une valeur t. Or, `find_offers`, qui se contente d'écrire dans les objets partagés, ne publie aucune valeur et `timer(2000)` publie un signal après 2000 millisecondes. La ligne 18 peut donc se lire : « exécuter `find_offers` pendant deux secondes puis passer à la ligne 19 ».

```
def find_best(agencies, destination) =
  -- sites auxiliaires internes
  def find_offers() =
    each(agencies) >agency> agency(destination) >offer>
      (offers.add(offer) |
      (best_offer.read() >o> compare(o, offer) >b> ift(b) >x>
        (best_agency.write(agency) | best_offer.write(offer))))
  #
  def extend_best() =
    best_agency.read() >ba> best_offer.read() >bo> ba.exists(bo) >b>
      (ift(b) >x> ba.get_info(bo) | iff(b) >x> alarm("inconsistent"))
  #
  def sort_offers() =
    offers.sort() ; best_offer.read() = offers.first() >b>
      (ift(b) >x> offers | iff(b) >x> alarm("not best"))
  #
  -- corps du programme
  ((t <t< (find_offers() | timer(2000))) >t>
    ((e_b, s_o) <e_b< extend_best() <s_o< sort_offers()))
  -- variables globales
  <offers< Stack()
  <best_offer< (Register() >r> r.write(null) ; r)
  <best_agency< Register()
#
```

(a) Identification de la meilleure offre pour une destination parmi celles proposées par des agences de voyage.

1. each([A1, A2])
2. timer(2000)
3. Register()
4. Register()
5. A1(D)
6. r.write(null)
7. best_offer.read()
8. Stack()
9. offers.add(O1)
10. A2(D)
11. offers.add(O2)
12. compare(null, 01)
13. best_offer.read()
14. compare(null, 02)
15. ift(true)
16. ift(true)
17. best_offer.write(O2)
18. best_offer.write(O1)
19. best_agency.write(A1)
20. best_agency.write(A2)
21. best_agency.read()
22. best_offer.read()
23. A2.exists(O1)
24. iff(false)
25. ift(false)
26. alarm("inconsistent")
27. offers.sort()
28. best_offer.read()
29. offers.first()
30. =(O1, O2)
31. iff(false)
32. ift(false)
33. alarm("not best")

(b) Une histoire séquentielle possible où $agencies = [A1, A2]$ et $destination = D$. Chaque agence publie une offre, respectivement $O1$ et $O2$. Pour des raisons de place et de clarté, nous ne représentons que les appels de sites externes dans cette exécution.

FIGURE 7.4 – Exemple de programme Orc.

Celle ci sert à construire la paire à publier, `extend_best` et `sort_offers` calculant chacun une composante en parallèle.

Le site `find_offers` commence par demander à toutes les agences les offres disponibles pour la destination donnée en paramètre (ligne 4) puis, pour chaque réponse, il insère l'offre dans la liste des offres (ligne 5) et la compare à la meilleure offre connue jusque-là. Le cas échéant, il met à jour les registres `best_offer` et `best_agency` (lignes 6 et 7).

Le site `extend_best` vérifie que l'offre contenue dans `best_offer` est bien proposée par l'agence contenue dans `best_agency`. Si c'est le cas, il demande à l'agence des informations complémentaires sur l'offre. Sinon, il lève une alarme `"inconsistent"`.

Enfin, le site `sort_offers` trie les offres présentes dans `offers` et vérifie que la meilleure offre est bien celle contenue dans `best_offer`. Sinon, il lève également une alarme `"not best"`.

La figure 7.4b montre une histoire séquentielle, c'est-à-dire une trace d'exécution possible pour le programme de la figure 7.4a, restreinte aux seuls appels de sites externes pour en faciliter la lecture. Dans cet exemple, les deux alarmes ont été levées en raison d'incohérences dans les registres partagés. Ces incohérences ont en fait été provoquées par l'édition concurrente des deux registres partagés pendant l'exécution de `find_offers`. On peut en effet remarquer que les deux offres trouvées sont comparées à la valeur du registre `best_offer` aux événements numérotés 7 et 13, qui surviennent tous les deux avant les écritures (événements 17 et 18). Ainsi, les deux lectures ont retourné la valeur par défaut (**null**) et les deux offres n'ont pas été comparées entre elles.

Une façon de régler le problème serait de remplacer le couple de registres par une structure de données adaptée rendant atomique le test et l'édition des deux registres. Encore faut-il être capable de détecter le moment de l'exécution où la première erreur s'est produite. En pratique, ces erreurs peuvent survenir bien avant leur détection. Or l'histoire séquentielle brute définie par la sémantique standard ne fournit aucune information sur la concurrence, ce qui rend impossible la détection des causes racines. Même en sachant que de nombreux bogues des applications réparties sont dus à des situations de compétition, l'absence d'information sur la concurrence rend la détection de celles-ci impossible.

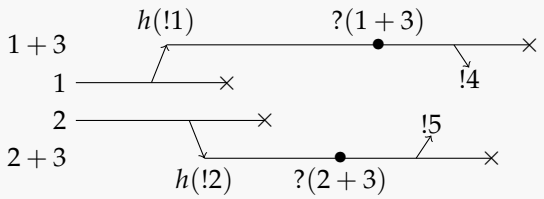
7.2 La sémantique instrumentée

7.2.1 Histoires concurrentes

Le principe d'une sémantique opérationnelle est d'expliciter la succession des événements qui peuvent se produire pendant une exécution, sous la forme d'étiquettes sur des transitions. L'ordre total sur les événements n'est pas pertinent pour des programmes répartis. Notre approche n'affecte pas la façon de générer les événements, mais ne tient pas compte de l'ordre dans lequel ils sont générés. À la place, nous ajoutons aux étiquettes l'information nécessaire à la reconstruction des relations d'ordre de processus et de programme entre les événements.

Plus précisément, une étiquette dans la sémantique instrumentée est un quadruplet $e = (e_k, e_l, e_c, e_a)$, où

- e_k est un identifiant, unique pour toute l'histoire, pris dans un ensemble dénombrable K,



(a) $(1 \mid 2) >x> (x + 3)$

La règle SEQV *de la sémantique standard (figure 7.5b) s'applique à une expression du type $f >x> g$ quand f publie une valeur v. Dans ce cas, une nouvelle instance de g est créée, dans laquelle les occurrences libres de x sont remplacées par la valeur v ($[v/x]g$), lancée en parallèle du programme $f' >x> g$ qu'il reste à exécuter. La figure 7.5a illustre la séquentialité et la concurrence mises en jeux par l'opérateur de séquentialité. L'expression $(1 \mid 2)$ publie les deux valeurs 1 et 2, donc dans le programme $(1 \mid 2) >x> (x + 3)$, le site $+$ est appelé deux fois. Ces deux appels sont concurrents entre eux, mais chaque appel est un successeur dans l'ordre de programme de l'une des publications. Plus précisément, tous les événements générés par l'expression $[v/x]g$ nouvellement créée ne peuvent se produire qu'après l'application de la règle* SEQV*. Par conséquent, cette expression est remplacée par $\langle [v/x]g, c \cup \{k\}, a \rangle_l$ dans la sémantique instrumentée (figure 7.5c). Ainsi, quels que soient les événements produits par $[v/x]g$ (l en indice représente tous les événements), on leur ajoutera comme antécédents l'application k de la règle* SEQV *et tous ses antécédents.*

$$\frac{f \xrightarrow{!v} f'}{f >x> g \xrightarrow{h(!v)} (f' >x> g) \mid [v/x]g}$$

(b) Sémantique standard

$$\frac{f \xrightarrow{k,!v,c,a}_i f'}{f >x> g \xrightarrow{k,h(!v),c,a}_i (f' >x> g) \mid \langle [v/x]g, c \cup \{k\}, a \rangle_l}$$

(c) Sémantique instrumentée

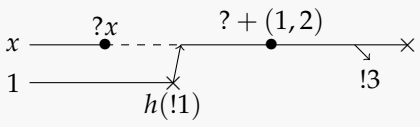
FIGURE 7.5 – Instrumentation de la règle SEQV

- e_l est une étiquette similaire à celles de la sémantique standard
- e_c contient l'ensemble fini des identifiants des événements de $\lfloor e \rfloor_{\mapsto}$
- e_a contient l'ensemble fini des identifiants des événements de $\lfloor e \rfloor_{\nearrow}$

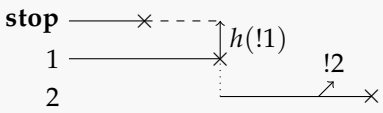
Les règles de la sémantique standard sont modifiées pour produire les informations supplémentaires. Leur version instrumentée est donnée sur la figure 7.8. Les règles les plus intéressantes sont détaillées dans les figures 7.5, 7.6 et 7.7. Remarquons que la récursivité décrit de la séquentialité entre les appels successifs du site récursif, et ne posent donc pas de problème particuliers pour l'instrumentation de la sémantique.

Une nouvelle construction syntaxique est nécessaire pour enregistrer le passé d'une expression : $\langle f, c, a \rangle_L$ signifie que c et a font partie du passé de l'expression instrumentée f. Ainsi, si f peut évoluer en f', cette transition doit également avoir c et a dans ses passés. L'indice L exprime le type d'événements activés par cette règle : $!v$ signifie que les causes ne s'ajoutent qu'à une publication, l que la règle est toujours valide et ω que la règle ne s'applique que lors de la terminaison du programme. Cette nouvelle construction est formellement définie par l'ajout de deux nouvelles règles dans la sémantique : CAUSEYES et CAUSENO. La syntaxe est seulement enrichie des constructions $\langle f, K, K \rangle_L$ dans les expressions et $\langle p, K, K \rangle_L$ dans les paramètres. L'ensemble des expressions autorisées par cette syntaxe étendue est noté Orc_i.

La sémantique instrumentée est également une sémantique opérationnelle. Comme toute sémantique opérationnelle, son ensemble de règles définit un système de transitions $\rightarrow_i$ et une sémantique séquentielle $[\![\cdot]\!]_i$. À partir d'une histoire séquentielle, on est capable de reconstruire une *histoire concurrente*, selon la définition 7.1.



(a) $x + 2 <x< 1$



(b) $(\textbf{stop} <x< 1)\ ;2$

La règle PRUNEV *de la sémantique standard (figure 7.6c) indique comment l'expression $f <x< g$ doit évoluer quand l'expression g publie une valeur v. Dans ce cas, toutes les occurrences libres de la variable x dans f sont remplacées par la valeur v et la partie de g qui reste à exécuter est préemptée. L'expression restante est donc $[v/x]f$. Deux sortes de relations sont induites par l'exécution de cette règle, qui se reflètent par une double instrumentation (figure 7.6d).*

- *La première dépendance est similaire à celle de la règle* SEQV *(figure 7.5) : la valeur publiée est un vecteur de séquentialité entre les deux expressions. On peut voir sur la figure 7.6a que dans l'expression $x + 2 <x< 1$, la publication cachée du 1 précède toujours l'appel du site $+$ et la publication du 3. L'événement correspondant à l'appel de la règle* PRUNEV *(étiqueté $h(!1)$) est un prédécesseur, selon l'ordre de processus, de tous les événements qui utilisent la valeur publiée (dont l'appel de $+$). Dans la sémantique instrumentée, ces relations sont capturées par le remplacement de v par $\langle v, c \cup \{k\}, a\rangle_l$.*
- *Dans l'expression $(\textbf{stop} <x< 1)\ ;2$ présentée sur la figure 7.6b, la publication 1 n'est liée à aucune variable de* **stop**, *donc l'instrumentation précédente n'est pas appliquée. Cependant, la publication du 2 ne peut avoir lieu qu'après la terminaison de $(\textbf{stop} <x< 1)$, qui ne peut se faire qu'après la publication du 1. Pour cette raison, l'ensemble du membre droit de la conclusion de la règle est également instrumentée par $\langle [v/x]f, c \cup \{k\}, a\rangle_\omega$. Notons l'indice ω qui indique que ces relations ne prennent effet qu'à la terminaison de $[v/x]f$. Ce cas illustre une difficulté de l'instrumentation de la sémantique. S'il existait une règle autorisant à terminer $f <x< g$ quand f est terminé, cette deuxième instrumentation ne serait pas nécessaire. L'instrumentation d'une règle ne dépend donc pas uniquement de cette règle, mais également des autres règles de la sémantique.*

$$\frac{g \xrightarrow{!v} g'}{f <x< g \xrightarrow{h(!v)} [v/x]f}$$

(c) Sémantique standard

$$\frac{g \xrightarrow{k,!v,c,a}_i g'}{f <x< g \xrightarrow{k,h(!v),c,a}_i \langle [\langle v, c \cup \{k\}, a\rangle_l / x]f, c \cup \{k\}, a\rangle_\omega}$$

(d) Sémantique instrumentée

FIGURE 7.6 – Instrumentation de la règle PRUNEV

Définition 7.1 (histoire concurrente)**.** Soit σ une histoire séquentielle obtenue en exécutant un programme f_0 grâce à la sémantique instrumentée :

$$\sigma = (e_k^1, e_l^1, e_c^1, e_a^1) \dots (e_k^n, e_l^n, e_c^n, e_a^n) \dots \in [\![f_0]\!]_i.$$

L'*histoire concurrente* correspondant à σ est le quintuplet $\overline{\overline{\sigma}} = (\Sigma, E, \Lambda, \mapsto, \nearrow)$ où :

- Σ est l'ensemble des étiquettes de la sémantique standard présentes dans σ :

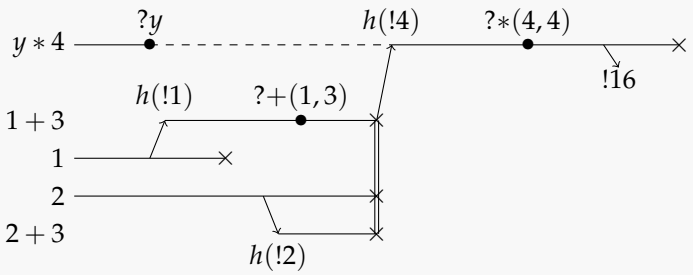
$$\Sigma = \{e_l^1, \dots, e_l^n, \dots\}$$

- $E = \{e_k^1, \dots, e_k^n, \dots\}$ est l'ensemble des identifiants dans l'histoire séquentielle :

$$E = \{e_k^1, \dots, e_k^n, \dots\}$$

- L'étiquette d'un événement est celle qui est juxtaposée à son identifiant dans le pas correspondant dans l'histoire séquentielle :

$$\Lambda : \left\{ \begin{array}{ccc} E & \to & \Sigma \\ e_k^i & \mapsto & e_l^i \end{array} \right.$$



(a) $4 * y <y< ((1 \mid 2) >x> x + 3)$

La règle PRUNEN *est intéressante car elle génère de la préemption. La figure 7.7a montre une exécution possible pour le programme* $4 * y <y< ((1 \mid 2) >x> x + 3)$*. Le programme* $(1 \mid 2) >x> x + 3$ *peut publier les deux valeurs* 4 *et* 5*, comme sur la figure 7.5a. Dans cette exécution, la publication du* 4 *préempte l'appel du site* $+(2,3)$*, mais pas la publication cachée du* 2*, qui a eu lieu avant. Il est donc possible d'observer* $h(!2)$ *puis* $h(!4)$ *ou bien* $h(!4)$ *seul, mais pas* $h(!4)$ *suivi de* $h(!2)$*, ce qui est bien de la préemption. Plus précisément, si la sous-expression* g *de* $f <x< g$ *produit un événement interne* n *puis une publication* $!v$*, soit* n *et* v *sont séquentiellement ordonnés, soit* n *est préempté par* v*. Dans les deux cas,* n *précède* v *dans l'ordre de programme* $\nearrow$*. Cela est indiqué par le remplacement de* g' *par* $\langle g', \emptyset, \{k\}\rangle_{!v}$ *dans la sémantique instrumentée (figure 7.7c).*

$$\frac{g \xrightarrow{n} g'}{f <x< g \xrightarrow{n} f <x< g'}$$

(b) Sémantique standard

$$\frac{g \xrightarrow{k,n,c,a}_i g'}{f <x< g \xrightarrow{k,n,c,a}_i f <x< \langle g', \emptyset, \{k\}\rangle_{!v}}$$

(c) Sémantique instrumentée

FIGURE 7.7 – Instrumentation de la règle PRUNEN

- L'ordre de processus est défini d'après les identifiants contenus dans e_c :

$$e^i_k \mapsto e^j_k \text{ si } e^i_k \in e^j_c \text{ ou } i = j$$

- L'ordre de programmes est défini d'après les identifiants contenus dans e_a :

$$e^i_k \nearrow e^j_k \text{ si } e^i_k \in e^j_a$$

Pour que la définition 7.1 corresponde à des histoires concurrentes telles que définies dans le chapitre 2, il faut s'assurer que l'ordre de processus est un bel ordre, que l'ordre de processus est contenu dans l'ordre de programme et que le passé de chaque événement selon l'ordre de programme est acyclique (voir la définition 2.13). Ces propriétés sont assurées par la façon dont on construit les histoires séquentielles, grâce aux règles de la sémantique instrumentée.

La figure 7.9 illustre le fonctionnement de la sémantique instrumentée sur l'exemple du programme $(x \mid 3) <x< (1 \mid 2)$. L'exécution des règles sur la figure décrit l'histoire concurrente de la figure 7.9a dont les linéarisations sont énumérées sur la figure 7.9b.

$$(\textsc{Publish})\ \frac{}{v \xrightarrow{k,!v,\emptyset,\emptyset}_i \langle \mathbf{stop}, \{k\}, \emptyset\rangle_l}\ v \text{ liée}, k \text{ fraîche} \qquad k \text{ fraîche}\ \frac{}{\mathbf{stop} \xrightarrow{k,\omega,\emptyset,\emptyset}_i \bot}\ (\textsc{Stop})$$

$$(\textsc{ExtStop})\ \frac{P \xrightarrow{k,!V,c,a}_i P' \qquad p \xrightarrow{k',\omega,c',a'}_i p'}{P(p) \xrightarrow{k,\omega,c\cup c',a\cup a'}_i \bot} \qquad \frac{P \xrightarrow{k,\omega,c,a}_i \bot}{P(p) \xrightarrow{k,\omega,c,a}_i \bot}\ (\textsc{StopCall})$$

$$(\textsc{ExtCall})\ \frac{P \xrightarrow{k,!V,c,a}_i P' \qquad p \xrightarrow{k',!v,c',a'}_i p'}{P(p) \xrightarrow{k,?V_k(v),c\cup c',a\cup a'}_i \langle ?k, c\cup c'\cup\{k\}, a\cup a'\rangle_l}$$

$$(\textsc{DefDeclare})\ \frac{[D/y]f \xrightarrow{k,l,c,a}_i f'}{D\#f \xrightarrow{k,l,c,a}_i f'}\ D \text{ est } \mathbf{def}\ y(x) = g$$

$$(\textsc{IntCall})\ \frac{P \xrightarrow{k,!D,c,a}_i P'}{P(p) \xrightarrow{k,?D,c,a}_i \langle [D/y][p/x]g, c\cup\{k\}, a\rangle_l}\ D \text{ est } \mathbf{def}\ y(x) = g$$

$$(\textsc{ParLeft})\ \frac{f \xrightarrow{k,l,c,a}_i f'}{f \mid g \xrightarrow{k,l,c,a}_i f' \mid g}\ l \neq \omega \qquad j \text{ fraîche}\ \frac{?k \text{ receives } T(v)}{?k \xrightarrow{j,!v,\emptyset,\emptyset}_i \mathbf{stop}}\ (\textsc{ResT})$$

$$(\textsc{ParRight})\ \frac{g \xrightarrow{k,l,c,a}_i g'}{f \mid g \xrightarrow{k,l,c,a}_i f \mid g'}\ l \neq \omega \qquad j \text{ fraîche}\ \frac{?k \text{ receives } NT(v)}{?k \xrightarrow{j,!v,\emptyset,\emptyset}_i ?k}\ (\textsc{ResNt})$$

$$(\textsc{ParStop})\ \frac{f \xrightarrow{k,\omega,c,a}_i f' \qquad g \xrightarrow{k',\omega,c',a'}_i g'}{f \mid g \xrightarrow{k,\omega,c\cup c',a\cup a'}_i \bot} \qquad j \text{ fraîche}\ \frac{?k \text{ receives } Neg}{?k \xrightarrow{j,\omega,\emptyset,\emptyset}_i \bot}\ (\textsc{ResNeg})$$

$$(\textsc{OthV})\ \frac{f \xrightarrow{k,!v,c,a}_i f'}{f\ ;\ g \xrightarrow{k,!v,c,a}_i f'} \qquad \frac{f \xrightarrow{k,!v,c,a}_i f'}{f >x> g \xrightarrow{k,h(!v),c,a}_i (f' >x> g) \mid \langle [v/x]g, c\cup\{k\}, a\rangle_l}\ (\textsc{SeqV})$$

$$(\textsc{OthN})\ \frac{f \xrightarrow{k,n,c,a}_i f'}{f\ ;\ g \xrightarrow{k,n,c,a}_i f'\ ;\ g} \qquad \frac{f \xrightarrow{k,n,c,a}_i f'}{f >x> g \xrightarrow{k,n,c,a}_i f' >x> g}\ (\textsc{SeqN})$$

$$(\textsc{OthStop})\ \frac{f \xrightarrow{k,\omega,c,a}_i \bot}{f\ ;\ g \xrightarrow{k,h(\omega),c,a}_i \langle g, c\cup\{k\}, a\rangle_l} \qquad \frac{f \xrightarrow{k,\omega,c,a}_i \bot}{f >x> g \xrightarrow{k,\omega,c,a}_i \bot}\ (\textsc{SeqStop})$$

$$(\textsc{PruneV})\ \frac{g \xrightarrow{k,!v,c,a}_i g'}{f <x< g \xrightarrow{k,h(!v),c,a}_i \langle [\langle v, c\cup\{k\}, a\rangle_l / x]f, c\cup\{k\}, a\rangle_\omega}$$

$$(\textsc{PruneN})\ \frac{g \xrightarrow{k,n,c,a}_i g'}{f <x< g \xrightarrow{k,n,c,a}_i f <x< \langle g', \emptyset, \{k\}\rangle_{!v}}$$

$$(\textsc{PruneLeft})\ \frac{f \xrightarrow{k,l,c,a}_i f'}{f <x< g \xrightarrow{k,l,c,a}_i f' <x< g}\ l \neq \omega$$

$$(\textsc{PruneStop})\ \frac{g \xrightarrow{k,\omega,c,a}_i \bot}{f <x< g \xrightarrow{k,h(\omega),c,a}_i \langle [\langle \mathbf{stop}, c\cup\{k\}, a\rangle_l / x]f, c\cup\{k\}, a\rangle_\omega}$$

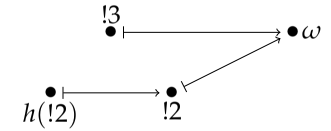
$$(\textsc{CauseYes})\ \frac{f \xrightarrow{k,l,c,a}_i f'}{\langle f, c', a'\rangle_L \xrightarrow{k,l,c\cup c',a\cup a'\cup c'}_i \langle f', c', a'\rangle_L}\ l \in L$$

$$(\textsc{CauseNo})\ \frac{f \xrightarrow{k,l,c,a}_i f'}{\langle f, c', a'\rangle_L \xrightarrow{k,l,c,a}_i \langle f', c', a'\rangle_L}\ l \notin L$$

FIGURE 7.8 – La version instrumentée des règles de la sémantique opérationnelle.

$$
\begin{array}{l}
(x \mid 3) <x< (1 \mid 2) \\
\quad \downarrow (1, !3, \emptyset, \emptyset) \\
(x \mid \langle \mathbf{stop}, \{1\}, \emptyset \rangle_l) <x< (1 \mid 2) \\
\quad \downarrow (2, h(!2), \emptyset, \emptyset) \\
\langle \langle 2, \{2\}, \emptyset \rangle_l \mid \langle \mathbf{stop}, \{1\}, \emptyset \rangle_l), \{2\}, \emptyset \rangle_\omega \\
\quad \downarrow (3, !2, \{2\}, \{2\}) \\
\langle \langle \langle \mathbf{stop}, \{3\}, \emptyset \rangle_l, \{2\}, \emptyset \rangle_l \mid \langle \mathbf{stop}, \{1\}, \emptyset \rangle_l, \{2\}, \emptyset \rangle_\omega \\
\quad \downarrow (4, \omega, \{1,2,3\}, \{1,2,3\}) \\
\langle \bot, \{2\}, \emptyset \rangle_\omega
\end{array}
$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{3 \xrightarrow{1,!3,\emptyset,\emptyset}_i \langle \mathbf{stop}, \{1\}, \emptyset \rangle_l}\,(\text{Publish})}{x \mid 3 \xrightarrow{1,!3,\emptyset,\emptyset} x \mid \langle \mathbf{stop}, \{1\}, \emptyset \rangle_l}\,(\text{ParRight})}{(x \mid 3) <x< (1 \mid 2) \xrightarrow{1,!3,\emptyset,\emptyset} (x \mid \langle \mathbf{stop}, \{1\}, \emptyset \rangle_l) <x< (1 \mid 2)}\,(\text{PruneLeft})}{\dfrac{\dfrac{\dfrac{}{2 \xrightarrow{2,!2,\emptyset,\emptyset}_i \langle \mathbf{stop}, \{2\}, \emptyset \rangle_l}\,(\text{Publish})}{1 \mid 2 \xrightarrow{2,!2,\emptyset,\emptyset} 1 \mid \langle \mathbf{stop}, \{2\}, \emptyset \rangle_l}\,(\text{ParRight})}{(x \mid \langle \mathbf{stop}, \{1\}, \emptyset \rangle_l) <x< (1 \mid 2) \xrightarrow{2,h(!2),\emptyset,\emptyset} \langle \langle 2, \{2\}, \emptyset \rangle_l \mid \langle \mathbf{stop}, \{1\}, \emptyset \rangle_l, \{2\}, \emptyset \rangle_\omega}\,(\text{PruneV})}{\dfrac{\dfrac{\dfrac{\dfrac{}{2 \xrightarrow{3,!2,\emptyset,\emptyset}_i \langle \mathbf{stop}, \{3\}, \emptyset \rangle_l}\,(\text{Publish})}{\langle 2, \{2\}, \emptyset \rangle_l \xrightarrow{3,!2,\{2\},\{2\}} \langle \langle \mathbf{stop}, \{3\}, \emptyset \rangle_l, \{2\}, \emptyset \rangle_l}\,(\text{CauseYes})}{\langle 2, \{2\}, \emptyset \rangle_l \mid \langle \mathbf{stop}, \{1\}, \emptyset \rangle_l \xrightarrow{3,!2,\{2\},\{2\}} \langle \langle \mathbf{stop}, \{3\}, \emptyset \rangle_l, \{2\}, \emptyset \rangle_l \mid \langle \mathbf{stop}, \{1\}, \emptyset \rangle_l}\,(\text{ParLeft})}{\langle \langle 2, \{2\}, \emptyset \rangle_l \mid \langle \mathbf{stop}, \{1\}, \emptyset \rangle_l, \{2\}, \emptyset \rangle_\omega \xrightarrow{3,!2,\{2\},\{2\}} \langle \langle \langle \mathbf{stop}, \{3\}, \emptyset \rangle_l, \{2\}, \emptyset \rangle_l \mid \langle \mathbf{stop}, \{1\}, \emptyset \rangle_l, \{2\}, \emptyset \rangle_\omega}\,(\text{CauseNo})}{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbf{stop} \xrightarrow{4,\omega,\emptyset,\emptyset} \bot}\,(\text{Stop})}{\langle \mathbf{stop}, \{3\}, \emptyset \rangle_l \xrightarrow{4,\omega,\{3\},\{3\}} \langle \bot, \{3\}, \emptyset \rangle_l}\,(\text{CauseYes})}{\langle \langle \mathbf{stop}, \{3\}, \emptyset \rangle_l, \{2\}, \emptyset \rangle_l \xrightarrow{4,\omega,\{2,3\},\{2,3\}} \langle \langle \bot, \{3\}, \emptyset \rangle_l, \{2\}, \emptyset \rangle_l}\,(\text{CauseYes}) \quad \dfrac{\dfrac{}{\mathbf{stop} \xrightarrow{5,\omega,\emptyset,\emptyset} \bot}\,(\text{Stop})}{\langle \mathbf{stop}, \{1\}, \emptyset \rangle_l \xrightarrow{5,\omega,\{1\},\{1\}} \langle \bot, \{1\}, \emptyset \rangle_l}\,(\text{CauseYes})}{\langle \langle \mathbf{stop}, \{3\}, \emptyset \rangle_l, \{2\}, \emptyset \rangle_l \mid \langle \mathbf{stop}, \{1\}, \emptyset \rangle_l \xrightarrow{4,\omega,\{1,2,3\},\{1,2,3\}} \bot}\,(\text{ParStop})}{\langle \langle \langle \mathbf{stop}, \{3\}, \emptyset \rangle_l, \{2\}, \emptyset \rangle_l \mid \langle \mathbf{stop}, \{1\}, \emptyset \rangle_l, \{2\}, \emptyset \rangle_\omega \xrightarrow{4,\omega,\{1,2,3\},\{1,2,3\}} \langle \bot, \{2\}, \emptyset \rangle_\omega}\,(\text{CauseYes})
$$



(a) Histoire concurrente

$$
\left\{
\begin{array}{ll}
h(!2) & !3 \quad h(!2) \cdot !2 \\
!3 \cdot h(!2) & h(!2) \cdot !3 \\
!3 \cdot h(!2) \cdot !2 & !3 \cdot h(!2) \cdot !2 \cdot \omega \\
h(!2) \cdot !3 \cdot !2 & h(!2) \cdot !3 \cdot !2 \cdot \omega \\
h(!2) \cdot !2 \cdot !3 & h(!2) \cdot !2 \cdot !3 \cdot \omega
\end{array}
\right\}
$$

(b) Linéarisations

FIGURE 7.9 – Exécution d'un programme Orc par la sémantique instrumentée.

7.2.2 Propriétés

Nous énonçons maintenant les principales propriétés vérifiées par la sémantique instrumentée. La proposition 7.1 justifie le nom de la sémantique instrumentée : si l'on ne conserve que le deuxième champ des étiquettes de la sémantique instrumentée, on obtient exactement la sémantique standard, ce qui implique les deux propriétés suivantes.

- Toutes les histoires concurrentes sont obtenues à partir d'une histoire séquentielle, que l'on peut obtenir à partir d'une simple projection.

- Toutes les histoires séquentielles peuvent être instrumentées de façon à obtenir une histoire concurrente. Il n'y a par contre pas unicité sur la façon de les instrumenter, signe que de l'information est perdue par la sémantique standard.

L'instrumentation ne fait donc bien que rajouter de l'information à la sémantique.

La proposition 7.2 prouve que l'instrumentation est correcte car elle ne définit aucun comportement incorrect. Si l'on part d'une exécution instrumentée, l'histoire concurrente correspondant produit plusieurs linéarisations. Toutes ces linéarisations sont elles-mêmes des histoires séquentielles acceptées par la sémantique standard. Une seule exécution avec la sémantique instrumentée produit donc autant d'information que plusieurs exécutions avec la sémantique standard. Nous ne donnerons pas ici les preuves complètes, qui sont uniquement des inductions énumérant tous les cas possibles. Elles peuvent être trouvées dans [90].

Proposition 7.1 (Instrumentation). *Soit* $\pi_l\left([\![f]\!]_i\right) = \{\sigma[1]_l...\sigma[n]_l... : \sigma[1]...\sigma[n]... \in [\![f]\!]_i\}$ *l'ensemble des projections des exécutions instrumentées sur leurs étiquettes. On a :*

$$\forall f \in \mathcal{O}, \pi_l\left([\![f]\!]_i\right) = [\![f]\!]$$

La preuve de cette proposition est simple puisque les deux sémantiques contiennent des règles similaires. L'arbre de dérivation de chaque pas a une structure similaire dans les deux sémantiques. Pour prouver cette propriété, il suffit donc de définir deux projections.

- Une projection remplace les occurrences de $\langle f, c', a' \rangle_L$ par f dans les expressions de Orc_i pour obtenir des expressions de la syntaxe standard.

- Une projection transforme les arbres de dérivation de la sémantique instrumentée en arbres de la sémantique standard. Cette projection supprime simplement les applications des règles CAUSEYES et CAUSENO.

Il suffit alors de montrer que pour toute expression de la sémantique instrumentée, il est équivalent de construire un pas d'exécution avec la sémantique instrumentée puis d'appliquer ces projections à l'arbre de dérivation et à l'expression obtenue, ou bien d'appliquer en premier lieu la projection à l'expression instrumentée, puis d'appliquer un pas de la sémantique standard. En appliquant ces deux projections à l'histoire calculée sur la figure 7.9, on obtient exactement l'histoire de la figure 7.3.

Proposition 7.2 (Correction). *Toutes les linéarisations d'une histoire concurrente obtenue par la sémantique instrumentée sont correctes pour la sémantique standard :*

$$\forall f \in Orc, \forall \alpha \in [\![f]\!]_i, \text{lin}\left(\overline{\overline{\alpha}}\right) \subset [\![f]\!]$$

Cette preuve est beaucoup plus compliquée. Prenons une linéarisation $L \in \text{lin}\left(\overline{\overline{\sigma}}\right)$. Pour prouver que $L \in [\![f]\!]$, nous devons montrer qu'il est possible de transformer progressivement σ en L en appliquant une succession de pas qui correspondent soit à l'inversion de deux événements consécutifs, soit à la préemption d'un événement par son successeur, soit à la prise du préfixe de la séquence. Comme $\sigma \in \text{lin}\left(\overline{\overline{\sigma}}\right)$ et chaque pas préserve la propriété, on obtient finalement $L \in \text{lin}\left(\overline{\overline{\sigma}}\right)$. La principale difficulté concerne le raisonnement sur les deux premiers types de pas, puisqu'une preuve est nécessaire pour chaque couple de règles successives possible.

7.3 Application

Replaçons nous dans le contexte de l'exemple de l'algorithme de la figure 7.4a présenté dans la section 7.1. Conformément à la proposition 7.1, l'exécution de la figure 7.4b peut également se produire dans la sémantique instrumentée. La figure 7.10 montre la même trace, mais cette fois avec l'information calculée par la sémantique instrumentée. La figure 7.11 montre l'histoire concurrente correspondant sous sa forme graphique. Si l'on suppose que les objets partagés sont au moins faiblement causalement cohérents (voir chapitre 4), il existe des relations causales entre les appels des opérations des objets partagés en plus de l'ordre de processus, représentées en pointillés sur la figure 7.11.

On aperçoit maintenant distinctement la structure de l'histoire, séparée en l'exécution des trois sites internes auxiliaires. L'exécution de `find_offers` est divisée en deux branches similaires, signe que deux offres ont été trouvées. Aucun relation de l'ordre de processus ne lie deux sites distincts. Cela est dû au fait que les sites communiquent seulement par objets partagés, comme dans les systèmes formés de processus communicant par messages. L'exécution de `find_offers` est préemptée par celle des deux autres sites, ce qui résulte en des liens dans l'ordre de programmes.

Analyse de causes racine. L'exécution instrumentée lève les deux mêmes alarmes que l'exécution séquentielle. Considérons leurs dernières causes communes, c'est-à-dire les événements qui sont dans le passé causal des deux alarmes mais dont les successeurs ne sont pas causes d'au moins une alarme. Les dernières causes communes sont au nombre de deux, étiquetées par `timer(2000)` et `best_offer.write(O1)`. Le minuteur n'est pas en cause ici, bien qu'une interruption entre les événements étiquetés par `best_offer.write(O1)` et `best_agency.write(A1)` pourrait également causer le même type d'erreurs. En fait, `best_offer.write(O1)` est bien la raison pour laquelle les deux alarmes ont été levées dans cette exécution. Si l'événement n'existait pas, la valeur publiée par `best_offer.read()` serait `O2` pour les deux appels, `A2.exists(O2)` et `=(O2, O2)` seraient vraies et les alarmes ne seraient pas levées.

Détection des situations de compétition. Les événements `best_offer.write(O1)` et `best_offer.write(O2)` sont concurrents, de même que `best_agency.write(O1)` et `best_agency.write(O2)`. Dans ce contexte, les événements peuvent s'entrelacer de sorte que `best_offer` et `best_agency` prennent des valeurs incohérentes. C'est ce qui s'est passé dans cette exécution.

1. (1, each([A1, A2]), ∅, ∅)
2. (2, timer(2000), ∅, ∅)
3. (3, Register(), ∅, ∅)
4. (4, Register(), ∅, ∅)
5. (5, A1(D), {1}, {1})
6. (6, r.write(null), {4}, {4})
7. (7, best_offer.read(), {1, 4, 5, 6}, {1, 4, 5, 6})
8. (8, Stack(), ∅, ∅)
9. (9, offers.add(O1), {1, 5, 8}, {1, 5, 8})
10. (10, A2(D), {1}, {1})
11. (11, offers.add(O2), {1, 8, 10}, {1, 8, 10})
12. (12, compare(null, 01), {1, 4, 5, 6, 7}, {1, 4, 5, 6, 7})
13. (13, best_offer.read(), {1, 4, 6, 10}, {1, 4, 6, 10})
14. (14, compare(null, 02), {1, 4, 6, 10, 13}, {1, 4, 6, 10, 13})
15. (15, ift(true), {1, 4, 5, 6, 7, 12}, {1, 4, 5, 6, 7, 12})
16. (16, ift(true), {1, 4, 6, 10, 13, 14}, {1, 4, 6, 10, 13, 14})
17. (17, best_offer.write(O2), {1, 4, 6, 10, 13, 14, 16}, {1, 4, 6, 10, 13, 14, 16})
18. (18, best_offer.write(O1), {1, 4, 5, 6, 7, 12, 15}, {1, 4, 5, 6, 7, 12, 15})
19. (19, best_agency.write(A1), {1, 3, 4, 5, 6, 7, 12, 15}, {1,3, 4, 5, 6, 7, 12, 15})
20. (20, best_agency.write(A2), {1, 3, 4, 6, 10, 13, 14, 16}, {1, 3, 4, 6, 10, 13, 14, 16})
21. (21, best_agency.read(), {2, 3}, {1, 2, 3})
22. (22, best_offer.read(), {2, 3, 4, 6, 21}, {1, 2, 3, 4, 6, 21})
23. (23, A2.exists(O1), {2, 3, 4, 6, 21, 22}, {1, 2, 3, 4, 6, 21, 22})
24. (24, iff(false), {2, 3, 4, 6, 21, 22, 23}, {1, 2, 3, 4, 6, 21, 22, 23})
25. (25, ift(false), {2, 3, 4, 6, 21, 22, 23}, {1, 2, 3, 4, 6, 21, 22, 23})
26. (26, alarm(“inconsistent”), {2, 3, 4, 6, 21, 22, 23, 24}, {1, 2, 3, 4, 6, 21, 22, 23, 24})
27. (27, offers.sort(), {2}, {2, 1})
28. (28, best_offer.read(), {2, 4, 6, 27}, {1, 2, 4, 6, 27})
29. (29, offers.first(), {2, 8, 27}, {1, 2, 8, 27})
30. (30, =(O1, O2), {2, 4, 6, 8, 27, 28, 29}, {1, 2, 4, 6, 8, 27, 28, 29})
31. (31, iff(false), {2, 4, 6, 8, 27, 28, 29, 30}, {1, 2, 4, 6, 8, 27, 28, 29, 30})
32. (32, ift(false), {2, 4, 6, 8, 27, 28, 29, 30}, {1, 2, 4, 6, 8, 27, 28, 29, 30})
33. (33, alarm(“not best”), {2, 4, 6, 8, 27, 28, 29, 30, 31}, {1, 2, 4, 6, 8, 27, 28, 29, 30, 31})

FIGURE 7.10 – Exécution calculée par la sémantique instrumentée.

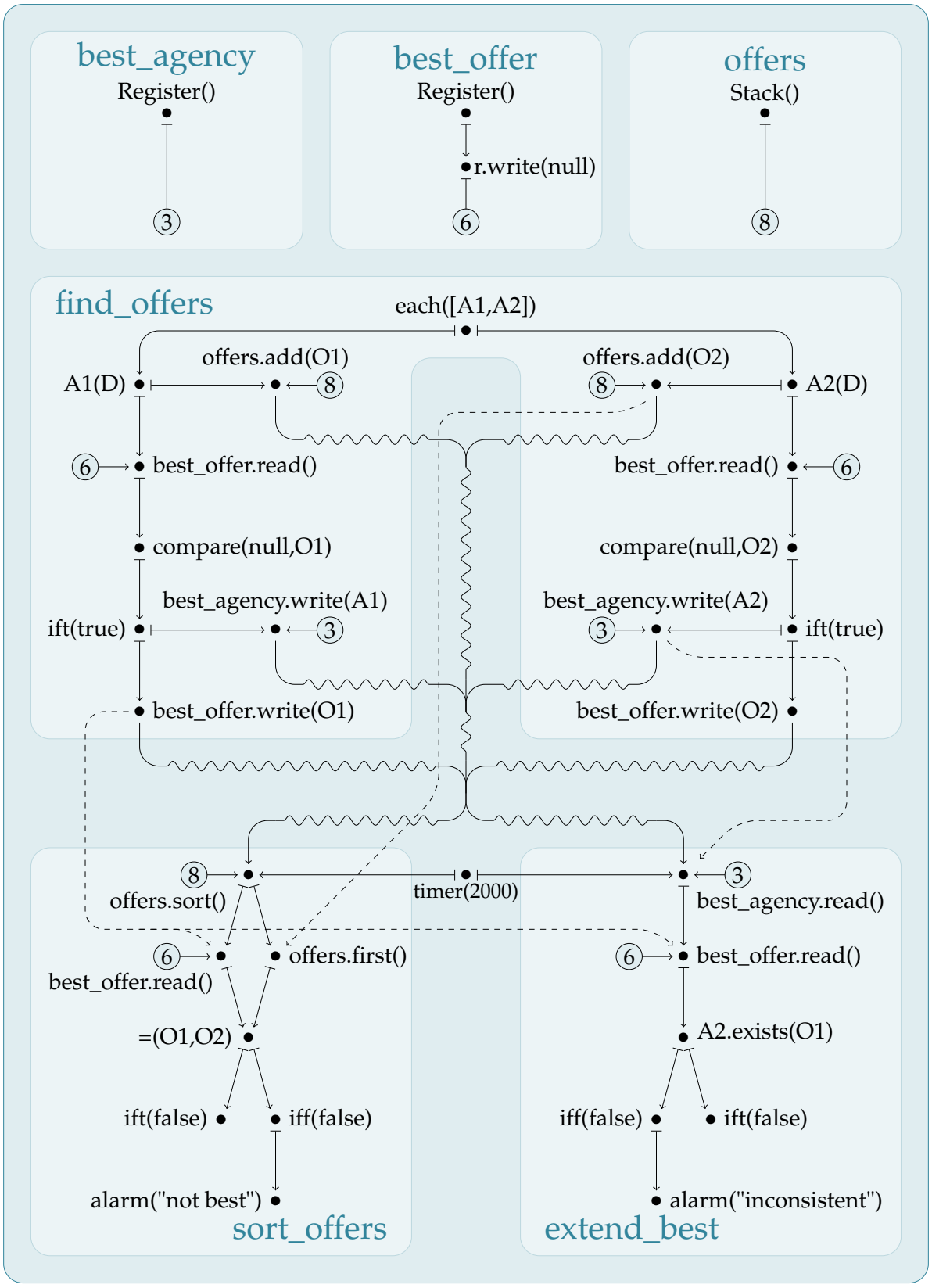


FIGURE 7.11 – Histoire concurrente dans sa forme graphique.

7.4 Travaux connexes

L'intérêt de suivre la trace des dépendances dynamiquement pendant l'exécution d'un programme pour contrôler l'exécution, détecter les erreurs et surveiller les performances est bien reconnu pour les applications réparties. Les techniques à base de vecteurs d'horloges, abondamment utilisées dans les systèmes répartis à passage de messages [42], ne suffisent pas dans le système plus complexe des orchestrations de service Web car un langage comme Orc peut produire des motifs de concurrence non-bornés. À notre connaissance, la seule instrumentation de ce type faite sur un programme [103] est basée sur du byte-code Java. Dans leur modèle, la seule source de causalité vient des variables partagées, ce qui ne correspond pas à des liens de la même nature que dans ce chapitre.

La seconde approche consiste à modifier la sémantique pour produire les informations de dépendance, ce qui mène à une sémantique concurrente. La difficulté est alors de conserver une forme d'équivalence avec la sémantique originale. Plusieurs techniques de débogage sont basées sur cette approche, comme [47] pour le langage Oz. Les principaux travaux dans le domaine des sémantiques concurrentes on été conduits sur les algèbres de processus, comme le π-calcul [24]. Notre contribution suit la même voie, mais pour le langage Orc et dans le contexte du calcul à grande échelle.

D'autres tentatives de sémantique concurrente pour Orc basées sur des structures d'événements ont été proposées. Dans [27], une sous-partie de Orc qui n'inclut pas la récursion utilise une connexion *ad hoc* avec les diagrammes de réseau de Petri ou avec le Join-calcul. Un autre encodage dans les réseaux de Petri colorés est proposé dans [100, 101] pour étudier la qualité de service. Enfin, une sémantique concurrente d'événements est proposée dans [102]. En pratique, ces sémantiques qui transforment à la compilation le code source en un modèle concurrent potentiellement infini sont difficiles à utiliser. Une sémantique instrumentée règle ce problème en permettant de calculer les dépendances à la volée pendant l'exécution.

Conclusion

Ce chapitre adresse le problème de la définition formelle d'une sémantique opérationnelle adaptée aux langages de programmation répartis. Nous avons basé notre travail sur une étude de cas, le langage Orc, car il est assez expressif pour produire de nombreuses situations présentes dans les systèmes répartis, dont la *séquentialité*, la *concurrence*, la *préemption* et le *conflit*, tout en restant assez simple pour que ce niveau de formalisme puisse y être appliqué. Notre contribution consiste en l'instrumentation de la sémantique opérationnelle structurelle standard de Orc. Cette instrumentation calcule des données supplémentaires relative à l'ordre de processus et à l'ordre de programme pendant l'exécution pour construire des histoires concurrentes. Cette information peut être utile pour accéder aux propriétés importantes des orchestrations. Nous illustrons ce point sur deux applications : l'analyse des *causes racine* et la détection des *situations de compétition*. Le débogage est donc une application directe. Au-delà du langage Orc, nous pensons que ce chapitre présente une approche générale qui peut être adaptée à d'autres langages non-déterministes où la concurrence est exprimée grâce à un jeu d'opérateurs.

En introduisant la concurrence et la préemption entre des événements qui étaient ordonnés arbitrairement par la sémantique standard, la sémantique instrumentée rassemble beaucoup d'histoires séquentielles en une seule histoire concurrente, ce qui

réduit grandement le nombre d'exécutions différentes. On peut voir sur la figure 7.9 que pour un programme aussi simple que $(x \mid 3) <x< (1 \mid 2)$, une seule histoire concurrente possède onze linéarisations, encodant autant d'exécutions séquentielles différentes. Une extension possible de ce travail consiste en l'élaboration d'une sémantique concurrente construisant une structure asymétrique d'événements représentant toutes les exécutions à la fois. L'intérêt — et la difficulté — d'une telle sémantique est qu'elle capture le conflit : la sémantique doit non seulement décrire *ce qui s'est passé*, mais également *ce qui aurait pu se passer*. Dans cet exemple, un 1 *et* un 2 sont publiés, chacun précédant l'autre dans l'ordre de programme. Deux événements en conflit ne peuvent pas se trouver dans la même linéarisation, donc la proposition 7.2 n'est pas mise en cause. En revanche, cela ne peut pas être obtenu grâce à une simple instrumentation de la sémantique standard (dans le sens de la proposition 7.1). Nous présentons une telle sémantique dans [90]. Son intérêt est de permettre le test des variantes possibles d'une exécution pour essayer de trouver des comportements anormaux.

8

Conclusion

Sommaire

8.1 Résumé

Les objets partagés sont centraux dans les systèmes répartis car ils peuvent modéliser toutes les couches des applications, depuis les primitives de communication jusqu'à l'application toute entière. Dans les systèmes asynchrones à passage de messages sans-attente, il est impossible d'implémenter des critères de cohérence forts comme la cohérence séquentielle ou la linéarisabilité : accepter que se produisent certaines incohérences est inévitable. Cela rend la spécification de ces objets et de leurs incohérences à la fois plus importante et plus compliquée. D'où la question soulevée par cette thèse : *comment spécifier les objets partagés d'un système sans-attente ?*

Nous affirmons que le meilleur moyen de spécifier les objets des systèmes sans-attente est le même que pour les objets fortement cohérents : la séparation de la *spécification séquentielle* qui décrit les aspects fonctionnels des opérations de l'objet et d'un *critère de cohérence* faible qui décrit les aspects liés à la qualité de service. Nous avons identifié dans l'introduction les trois qualités que devait avoir une bonne spécification. Il est maintenant temps d'y confronter nos travaux.

Rigueur. Le chapitre 2 présente le cadre mathématique dans lequel s'inscrit le reste du travail. La modélisation est bâtie autour de trois concepts centraux :

1. les *spécifications séquentielles* définies à l'aide de systèmes de transitions ;
2. les *histoires concurrentes*, des structures asymétriques d'événements étiquetées qui décrivent les relations de séquentialité, de concurrence, de préemption et de conflit entre les événements ;
3. les *critères de cohérence* qui associent un ensemble d'histoires cohérentes à chaque spécification séquentielle.

Tous les critères de cohérence présentés sont définis à l'aide de quelques opérateurs de base sur les histoires concurrentes, projections et linéarisations, dont la définition mathématique ne laisse pas place à l'interprétation.

Abstraction. Les critères de cohérence sont définis à partir des histoires concurrentes qui sont elles-mêmes des abstractions des systèmes sur lesquels les objets peuvent être implémentés. Les algorithmes et les preuves d'impossibilité que l'on donne tout au long de la thèse ne concernent que les systèmes sans-attente mais le chapitre 7 prouve que l'abstraction des histoires concurrentes est assez générale pour modéliser un système très différent : les orchestrations de services Web. Le langage Orc est décrit par une sémantique opérationnelle structurelle qui produit des exécutions séquentielles mal adaptées pour décrire la complexité des systèmes répartis. Nous avons proposé une instrumentation de cette sémantique qui calcule les informations relatives à la concurrence et la préemption pendant l'exécution pour produire des histoires concurrentes. Ainsi, tous les critères de concurrence définis pour les systèmes sans-attente ont également un sens pour les orchestrations écrites en Orc.

Simplicité. Séparer la spécification en deux facettes permet d'appréhender chacune avec deux fois moins de difficulté. Cela répond à la remarque de Shavit dans [113] : « il est infiniment plus simple et plus intuitif pour nous humains de spécifier comment les structures de données abstraites se comportent dans un cadre séquentiel. » La spécification séquentielle n'utilise que les concepts de système de transitions et de langage formel qui sont centraux dans toutes les branches de l'informatique. La complexité est donc entièrement reportée sur les critères de cohérence, dont chacun permet de spécifier une infinité d'objets différents. Le chapitre 6 illustre l'intérêt de se baser sur la spécification séquentielle. Il présente CODS, une bibliothèque pour le langage D qui met en œuvre les concepts défendus dans cette thèse : les spécifications séquentielles correspondent aux classes du langage orienté objets et les critères de cohérence sont fournis par la bibliothèque. L'utilisation de CODS est presque transparente : seule l'instanciation des objets est remplacée par une connexion à une instance partagée préexistante, par l'intermédiaire du critère de cohérence. La bibliothèque fournit également des techniques pour définir des transactions en dehors des spécifications séquentielles. Un cadre permet de définir de nouveaux critères de cohérence.

L'autre aspect de la problématique concerne la modélisation des systèmes sans-attente. Un critère de cohérence plus faible que la cohérence séquentielle et pour lequel il est possible d'implémenter tous les objets dans un système sans-attente est appelé un *critère faible*. Dans le chapitre 5, nous avons étudié l'espace des critères faibles comme un objet mathématique. En tant que cône tronqué, il ne possède pas de maximum. Nous avons identifié trois familles de critères faibles, dits *secondaires* qui, s'ils sont conjugués deux par deux, donnent un critère fort. Chaque critère secondaire possède un critère *primaire* opposé qui explicite le type d'incohérences qu'il admet.

Cohérence pipeline / convergence. Dans une histoire pipeline cohérente, les processus ne se soucient pas des lectures faites par les autres processus. Il en résulte que différents processus peuvent appliquer les opérations dans un ordre différent, ce qui empêche la convergence. La cohérence pipeline est surtout utile aux

objets utilisés par des programmes informatiques, comme les simulations scientifiques : chaque lecture est importante car elle sert aux calculs dans la suite du programme, mais la convergence importe peu, soit car l'exécution est censée terminer, soit car les processus n'arrêtent jamais d'écrire. Un autre exemple d'utilisation est la synchronisation entre les processus, par exemple pour filtrer l'entrée dans une section critique.

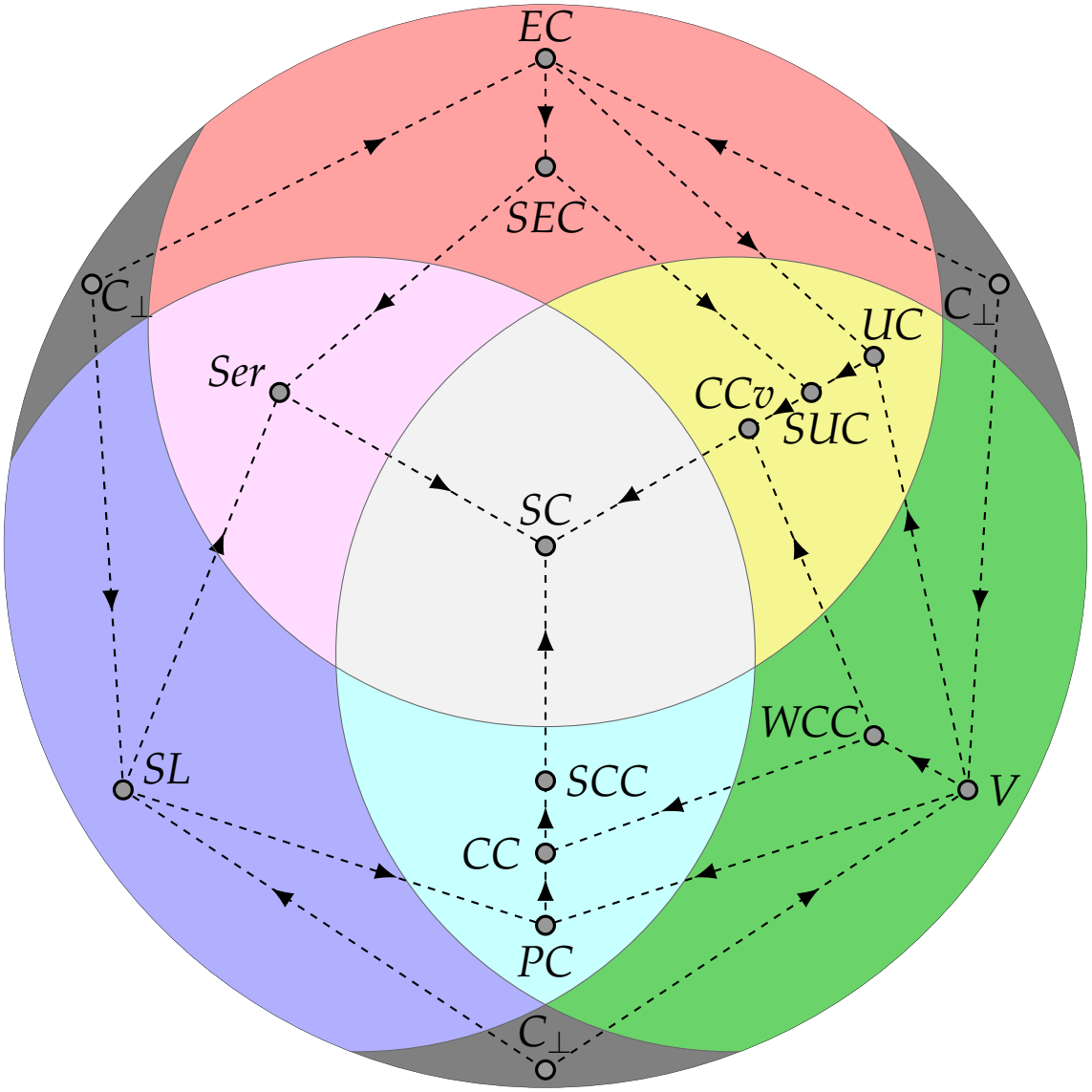
Cohérence d'écriture / localité d'état. La cohérence d'écritures renforce la convergence en forçant l'état de convergence à être l'un des états admissibles par la cohérence séquentielle. Le chapitre 3 étudie en détails ce critère et sa variante forte, qui peuvent tous les deux être implémentés dans les systèmes sans-attente. Pour garantir la convergence, la cohérence d'écritures est parfois obligée de revenir sur des choix, ce qui peut porter à violer des règles fixées par la spécification séquentielle (la localité d'états). Elle est donc plus adaptée à des applications collaboratives, car les utilisateurs humains peuvent s'adapter aux incohérences et les corriger, mais ils ont besoin de la convergence pour se mettre d'accord sur un document final.

Sérialisabilité / validité. Les opérations d'une histoire sérialisable peuvent être avortées ou acceptées. la sérialisabilité offre des garanties très fortes sur les opérations acceptées qui forment une histoire séquentiellement cohérente. En revanche, il n'y a pas de limite sur le nombre d'opérations avortées, donc elle n'assure pas qu'il sera toujours possible d'exécuter des opérations. De plus, pour diminuer le nombre d'opérations avortées, il peut être nécessaire de s'éloigner du modèle sans-attente. En résumé, la sérialisabilité est surtout à conseiller pour les applications pour lesquelles il est primordial d'éviter les erreurs quitte à empêcher momentanément une opération de s'effectuer.

En théorie, on pourrait construire une infinité de critères faibles impossibles à conjuguer deux à deux. Au-delà de cette remarque purement théorique, nous pensons que les trois critères secondaires ont une importance particulière. D'une part, ils correspondent chacun à une façon d'affaiblir l'une des propriétés du Consensus, qui est connu comme un problème fondamental de l'algorithmique du répartie. D'autre part, nous avons pu identifier des services de messagerie instantanée grand public dont les stratégies face à une déconnexion illustrent les trois types d'incohérence.

Chaque famille possède sa propre structure interne. Nous avons fait un petit pas dans leur exploration dans le chapitre 4 en étudiant la causalité. Celle-ci ne forme pas une famille à part, mais peut renforcer les autres critères au sein de leur famille. Nous avons défini la cohérence causale faible dans la famille de la validité, la convergence causale dans la famille de la cohérence d'écritures et les cohérences causale et causale forte dans la famille de la cohérence pipeline. La carte de la figure 8.1 résume les principaux critères étudiés dans cette thèse.

Enfin, l'étude des critères faibles est utile pour comprendre la calculabilité dans les systèmes répartis. En effet, la seule hiérarchie d'objets partagés connue à ce jour, la hiérarchie de Herlihy, est basée sur la capacité d'un objet à résoudre le Consensus. Nous avons montré qu'il existait un moyen d'affiner le premier niveau de la hiérarchie en faisant varier le critère de cohérence. Nous avons ainsi identifié la classe des CADT, pour lesquels la cohérence pipeline est plus forte que la convergence.



SC Cohérence séquentielle (p. 46)	*PC* Cohérence pipeline (p. 62)	*SEC* Convergence forte (p. 56)	*SUC* Cohérence d'écritures forte (p. 77)	*CCv* Convergence causale (p. 110)	*WCC* Cohérence causale faible (p. 108)	*SCC* Cohérence causale forte (p. 118)
$C_\perp$ Critère minimal (p. 40)	*SL* Localité d'état (p. 129)	*EC* Convergence (p. 54)	*UC* Cohérence d'écritures (p. 75)	*Ser* Sérialisabilité (p. 52)	*V* Validité (p. 128)	*CC* Cohérence causale (p. 113)

FIGURE 8.1 – Cartographie de l'espace des critères faibles.

8.2 Perspectives

De nombreuses questions ont été ouvertes par ce travail mais n'ont pas pu être traitées dans ce manuscrit. Nous en explicitons quelques-unes maintenant.

Cartographie de l'espace des critères faibles. Dans le chapitre 2, nous avons remarqué que la cohérence faible était souvent résumée par une hiérarchie de critères : la convergence est le plus faible, la cohérence pipeline le renforce en prenant en compte l'ordre des processus, puis vient l'ordre causal qui ajoute en plus les relations sémantiques entre les écritures et les lectures, et enfin la cohérence forte. L'étude approfondie de l'espace des critères faibles montre un visage bien différent. Pourrait-on transposer cette hiérarchie à l'intérieur de chaque famille ? Nous avons vu que la causalité était une composante structurante de plusieurs familles. Qu'en-est-il de l'ordre de processus ? Existe-t-il un critère intermédiaire entre la validité et la cohérence causale faible qui le prend en compte ? Comment se compose-t-il avec la cohérence d'écriture ? Inversement, peut-on le retirer de la cohérence pipeline ? Et si oui, le critère résultant est-il toujours un bon représentant de sa famille, ou au contraire, est-il possible d'implémenter la mémoire vérifiant à la fois la convergence et ce nouveau critère ? Outre la causalité et l'ordre FIFO, existe-t-il d'autres propriétés communes qui donnent une structure semblable à plusieurs familles ?

Dépendance aux systèmes. Bien que l'une de nos préoccupations était de s'abstraire au maximum du système sur lequel les algorithmes étaient implémentés, plusieurs résultats sont liés au système $AS_n[\emptyset]$. On peut se demander comment un changement de système les affecte.

D'un point de vue théorique, les familles de critères secondaires restent inconciliables tant que l'impossibilité du Consensus tient. En revanche, il n'est pas sûr que toutes puissent toujours y être implémentées. Si l'on autorise les processus à se rejoindre, ce qui se traduit par des événements ayant plusieurs antécédents incomparables dans l'ordre de processus (ce qui est le cas dans Orc), il est indispensable que tous les processus aient convergé avant la jonction pour que l'histoire soit pipeline cohérente. Dans ces systèmes, la cohérence pipeline est donc aussi difficile à implémenter que sa conjonction avec la convergence.

D'un point de vue pratique, les algorithmes mis en œuvre sont très différents d'un système à l'autre et donc leur complexité également. Par exemple, avec un temps maximal sur la durée d'acheminement des messages, on peut diminuer considérablement la complexité de la cohérence d'écritures. Nous avons comme projet d'adopter les algorithmes pour la cohérence d'écritures aux systèmes pair-à-pair.

Implémentations spécifiques. Dans cette thèse, nous nous sommes surtout concentrés sur des algorithmes génériques qui fonctionnent quelle que soit la spécification séquentielle passée en paramètre. Pour l'ensemble avec la cohérence d'écritures forte, nous avons vu qu'un algorithme dédié pouvait être beaucoup plus efficace. Il serait intéressant d'avoir des algorithmes optimisés pour d'autres types de données.

À terme, on pourrait rêver que CODS puisse trouver elle-même les meilleurs optimisations en fonction du code de la spécification séquentielle. Pour cela, des techniques d'analyse sémantique complexes doivent être mises en œuvre. La puissance d'introspection du langage D n'est probablement pas suffisante pour cela et un langage dédié est certainement nécessaire. En attendant, on pourrait ajouter des balises pour donner

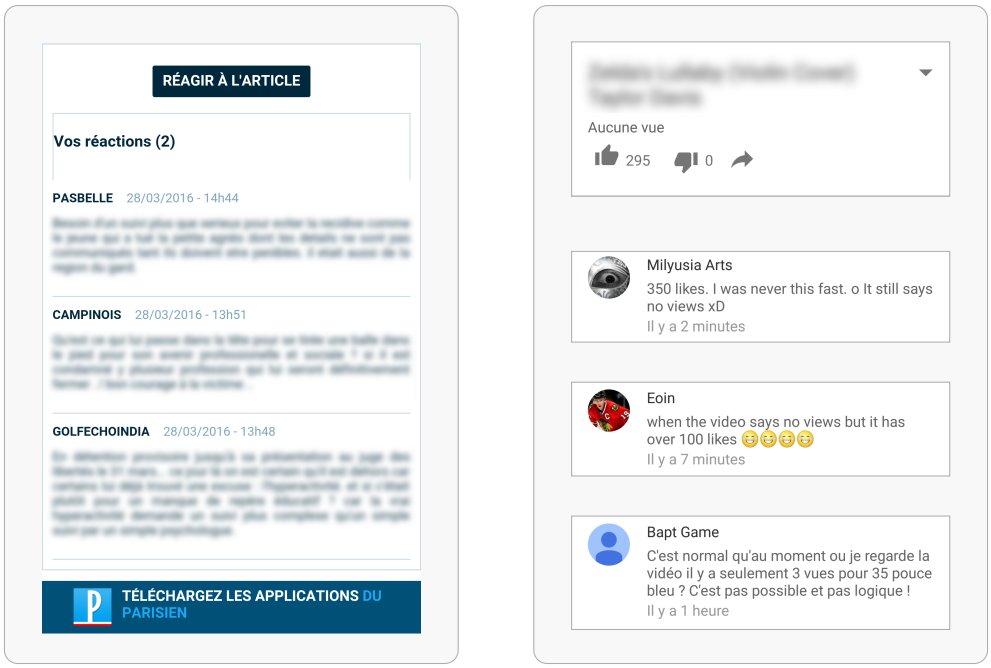


FIGURE 8.2 – Violation d'intégrité sur les sites du Parisien et de YouTube.

des informations à l'algorithme du critère de cohérence, comme `@update` et `@query` pour identifier les écritures et lectures pures.

Résolution des contraintes d'intégrité. L'absence de composabilité et le choix de critères de cohérence trop faibles peuvent mener à des lectures retournant des résultats incompréhensibles. La figure 8.2 montre deux illustrations de ce phénomène sur des sites Web. À gauche, la section des commentaires sous un article du quotidien français Le Parisien[1] indique que seulement deux commentaires ont été postés mais en affiche trois. À droite, il est fréquent que la plateforme d'hébergement de vidéos YouTube[2] indique un nombre de visionnages inférieur au nombre de votes sur le contenu, ce qui est étonnant compte tenu du fait que chaque membre ne peut voter qu'une seule fois par vidéo et qu'il est presque impossible de voter pour une vidéo sans être compté comme spectateur. Ces incohérences sont parfois remarquées par les visiteurs de ces sites, comme en attestent certains commentaires[3] sur YouTube.

Pour les bases de données, de telles incohérences sont regroupées sous le terme *contraintes d'intégrité*, que l'on peut définir comme des invariants sur l'ensemble des états accessibles du type de données abstraits. La plupart des critères de cohérence relient les valeurs lues à une linéarisation d'écritures, c'est-à-dire qu'ils imposent une forme d'accessibilité dans les types de données abstraits. Dans son stage de première année de Master, Damien Maussion a proposé une structure de données intégrant ces contraintes et vérifiant la cohérence d'écritures, d'abord en utilisant la bibliothèque CODS, puis en développant un algorithme dédié. Il a ainsi démontré qu'il était pos-

[1] http://www.leparisien.fr/
[2] https://www.youtube.com/
[3] Sur les images pour YouTube, les trois commentaires et les statistiques viennent de captures d'écran différentes (le commentaire en français a été posté sur une vidéo différente). Cependant, la page n'a pas été rechargée entre la capture des statistiques et celle du premier commentaire en anglais, ce qui montre l'absence de causalité sur le compteur de votes positifs, supposé croissant.

sible de vérifier les contraintes de domaine, les contraintes d'unicité et les contraintes référentielles (ce qui est suffisant pour implémenter les clés étrangères), sans synchronisation.

Critères hybrides. Nous avons vu dans le chapitre 2 que plusieurs critères de cohérence étaient des critères hybrides entre plusieurs autres critères faibles. Nous n'avons pour l'instant considéré que des critères définis directement, mais la question de leur hybridation est importante. Par exemple, comment se comportent deux objets vérifiant des critères différents au sein d'un même programme. Si l'un des critères est plus fort que l'autre, on aimerait que la composition vérifie au moins le critère le plus faible.

Une autre question est celle de deux opérations vérifiant des critères différents. Pour reprendre l'exemple des services de messagerie instantanée, la suppression d'un message est généralement locale : seul l'utilisateur qui supprime un message le voit disparaître de sa file de messages. Si l'on prend en compte cette nouvelle opération, la convergence ne peut être garantie que si aucun message n'est supprimé. Comment modéliser ce phénomène ?

Vers une nouvelle hiérarchie des objets séquentiels ? Nous avons vu que l'étude des critères faibles était un bon outil pour étudier la calculabilité des systèmes répartis. De façon plus générale, on peut se demander s'il existe une hiérarchie d'objets séquentiels qui les classe selon la difficulté à les implémenter dans un système sans-attente et, si elle existe, si elle est compatible avec la hiérarchie de Herlihy. Pour illustrer ces questions, nous proposons la conjecture 8.1.

Conjecture 8.1. *Soit $T \in \mathcal{T}_{UQ}$ un UQ-ADT tel que $(PC + EC)(T)$ peut être implémenté dans $AS_n[\emptyset]$. Alors $SC(T)$ peut être implémenté dans $AS_n[t < \frac{n}{2}]$.*

Si cette conjecture est vraie, nous avons déjà identifié deux catégories d'objets dont le nombre de Consensus est 1, définies à partir de critères de cohérence et dans des systèmes très différents, et qui affinent la hiérarchie de Herlihy. On sait que cette classe contient les CADT et le registre, mais pas la mémoire constituée de trois registres. Qu'en est-il de deux registres ?

Bibliographie

[1] Sarita V Adve and Hans-J Boehm. Memory models: a case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, 2010. 61

[2] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996. 61

[3] Sarita V Adve and Mark D Hill. Implementing sequential consistency in cache-based systems. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume 1, pages 47–50, 1990. 63

[4] Marcos K Aguilera, Svend Frolund, Vassos Hadzilacos, Stephanie L Horn, and Sam Toueg. Abortable and query-abortable objects and their efficient implementation. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 23–32. ACM, 2007. 53

[5] Mustaque Ahamad, Rida A Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 251–260. ACM, 1993. 64

[6] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995. 19, 20, 24, 33, 65, 111, 119

[7] Amitanand Aiyer, Lorenzo Alvisi, and Rida A Bazzi. On the availability of non-strict quorum systems. In *Distributed Computing*, pages 48–62. Springer, 2005. 50

[8] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *Proceedings 5th Biennial Conference on Innovative Data Systems Research*, pages 249–260, 2011. 57

[9] Peter Alvaro, William R Marczak, Neil Conway, Joseph M Hellerstein, and David Maier. Logic and Lattices for Distributed Programming, 2012. 57

[10] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, et al. Business process execution language for web services, 2003. 157

[11] Khaled Aslan, Pascal Molli, Hala Skaf-Molli, Stéphane Weiss, et al. C-Set: a commutative replicated data type for semantic stores. In *RED: Fourth International Workshop on REsource Discovery*, 2011. 57

[12] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995. 17, 49, 137

[13] Hagit Attiya, Faith Ellen, and Adam Morrison. Limitations of Highly-Available Eventually-Consistent Data Stores. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 385–394. ACM, 2015. 65

[14] Hagit Attiya and Jennifer L Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems (TOCS)*, 12(2):91–122, 1994. 17, 18, 45, 49, 123, 138

[15] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 22. ACM, 2012. 65

[16] Roberto Baldoni, Miroslaw Malek, Alessia Milani, and S Tucci Piergiovanni. Weakly-persistent causal objects in dynamic distributed systems. In *Reliable Distributed Systems, 2006. SRDS'06. 25th IEEE Symposium on*, pages 165–174. IEEE, 2006. 65, 117

[17] Roberto Baldoni, Alessia Milani, and Sara Tucci Piergiovanni. An optimal protocol for causally consistent distributed shared memory systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 68. IEEE, 2004. 65, 117

[18] Philip A Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983. 53

[19] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987. 52

[20] Brian N Bershad and Matthew J Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, 1991. 67

[21] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. *arXiv preprint arXiv:1210.3368*, 2012. 57, 58, 59, 60

[22] Kenneth P Birman and Thomas A Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1):47–76, 1987. 38

[23] Hans-J Boehm and Sarita V Adve. Foundations of the C++ concurrency memory model. In *ACM SIGPLAN Notices*, volume 43, pages 68–78. ACM, 2008. 15

[24] Michele Boreale and Davide Sangiorgi. A fully abstract semantics for causality in the π-calculus. In *STACS 95*, pages 243–254. Springer, 1995. 176

[25] Yuri Breitbart, Dimitrios Georgakopoulos, Marek Rusinkiewicz, and Abraham Silberschatz. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering*, 17(9):954–960, 1991. 53

[26] Eric A Brewer. Towards robust distributed systems. In *PODC*, page 7, 2000. 49, 123

[27] Roberto Bruni, Hernán Melgratti, and Emilio Tuosto. Translating Orc features into Petri nets and the Join calculus. In *Web Services and Formal Methods*, pages 123–137. Springer, 2006. 176

[28] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In *Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 271–284. ACM, 2014. 19, 55, 58, 61

[29] Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually consistent transactions. In *Programming Languages and Systems*, pages 67–86. Springer, 2012. 53

[30] Jacob Carlborg. Orange. `https://github.com/jacob-carlborg/orange`. Accessed: 2016-06-30. 153

[31] Armando Castaneda, Sergio Rajsbaum, and Michel Raynal. Specifying concurrent problems: beyond linearizability and up to tasks. In *Distributed Computing*, pages 420–435. Springer, 2015. 51

[32] Bernadette Charron-Bost. Concerning the size of clocks. In *Semantics of Systems of Concurrent Processes*, pages 176–184. Springer, 1990. 65

[33] The D community. The D Programming Language. `http://dlang.org/`. Accessed: 2016-06-30. 147

[34] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008. 63

[35] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007. 24, 53

[36] John Derrick, Brijesh Dongol, Gerhard Schellhorn, Bogdan Tofan, Oleg Travkin, and Heike Wehrheim. Quiescent consistency: Defining and verifying relaxed linearizability. In *FM 2014: Formal Methods*, pages 200–214. Springer, 2014. 50

[37] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)*, 34(1):77–97, 1987. 50

[38] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. *ACM SIGARCH Computer Architecture News*, 14(2):434–442, 1986. 65

[39] Swan Dubois, Rachid Guerraoui, Petr Kuznetsov, Franck Petit, and Pierre Sens. The weakest failure detector for eventual consistency. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 375–384. ACM, 2015. 51

[40] Azadeh Farzan and Parthasarathy Madhusudan. Causal atomicity. In *Computer Aided Verification*, pages 315–328. Springer, 2006. 65

[41] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems (TOCS)*, 19(2):171–216, 2001. 49

[42] Colin J Fidge. *Timestamps in message-passing systems that preserve the partial ordering*. Australian National University. Department of Computer Science, 1987. 176

[43] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985. 50, 131, 132

[44] Roy Friedman, Michel Raynal, and François Taïani. Fisheye consistency: Keeping data in synch in a georeplicated world. In *NETYS-3rd International Conference on NETwork sYStems*, 2015. 67

[45] Pranav Gambhire and Ajay D Kshemkalyani. Reducing false causality in causal message ordering. In *High Performance Computing—HiPC 2000*, pages 61–72. Springer, 2000. 117

[46] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. *Memory consistency and event ordering in scalable shared-memory multiprocessors*, volume 18. ACM, 1990. 63, 67

[47] Elena Giachino, Ivan Lanese, and Claudio Antares Mezzina. Causal-consistent reversible debugging. In *Fundamental Approaches to Software Engineering*, pages 370–384. Springer, 2014. 176

[48] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002. 18, 24, 49, 123, 125, 138

[49] James R Goodman. *Cache consistency and sequential consistency*. University of Wisconsin-Madison, Computer Sciences Department, 1991. 63, 64

[50] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184. ACM, 2008. 53

[51] Stefan Haar, Claude Jard, and Guy-Vincent Jourdan. Testing input/output partial order automata. In *Testing of Software and Communicating Systems*, pages 171–185. Springer, 2007. 51

[52] Vassos Hadzilacos. A theory of reliability in database systems. *Journal of the ACM (JACM)*, 35(1):121–145, 1988. 53

[53] Vassos Hadzilacos and Sam Toueg. Reliable Broadcast and Related Problems. *Distributed Systems*, pages 97–145, 2013. 37, 38

[54] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983. 51

[55] Jean-Michel Hélary and Alessia Milani. About the efficiency of partial replication to implement distributed shared memory. In *Parallel Processing, 2006. ICPP 2006. International Conference on*, pages 263–270. IEEE, 2006. 65, 117

[56] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991. 21, 50, 132, 134, 137

[57] Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993. 51

[58] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990. 48

[59] Phillip W Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 302–309. IEEE, 1990. 64

[60] Damien Imbs and Michel Raynal. Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theoretical Computer Science*, 444:113–127, 2012. 53

[61] Jean-Marc De Jaeger. Le Figaro. `http://www.lefigaro.fr/secteur/high-tech/2016/06/07/32001-20160607ARTFIG00208-une-faille-de-facebook-messenger-permettait-de-trafiquer-les-conversations.php`. Accessed: 2016-06-30. 71

[62] Ernesto Jiménez, Antonio Fernández, and Vicent Cholvi. A parametrized algorithm that implements sequential, causal, and cache memory consistencies. *Journal of Systems and Software*, 81(1):120–131, 2008. 67

[63] Karsenty, Alain and Beaudouin-Lafon, Michel. An algorithm for distributed groupware applications. In *ICDCS*, 1993. 79

[64] Bettina Kemme. One-copy-serializability. In *Encyclopedia of Database Systems*, pages 1947–1948. Springer, 2009. 53

[65] David Kitchin, William R Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. In *CONCUR 2006–Concurrency Theory*, pages 477–491. Springer, 2006. 158

[66] David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. ORC language. `http://orc.csres.utexas.edu`. Accessed: 2016-06-30. 21, 158

[67] David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. The Orc programming language. In *Formal Techniques for Distributed Systems*, pages 1–25. Springer, 2009. 158

[68] Martin Kleppmann. A Critique of the CAP Theorem. *arXiv preprint arXiv:1509.05393*, 2015. 49

[69] Hiromichi Kobashi, Yasuo Yamane, Miho Murata, Toshiaki Saeki, Hiroki Moue, and Yuichi Tsuchimoto. Eventually Consistent Transaction. In *Proceedings of the 22nd IASTED International Conference on Parallel and Distributed Computing and Systems, p103-109*, 2010. 53

[70] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010. 24, 53

[71] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. 38, 79

[72] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979. 42, 45, 46

[73] Leslie Lamport. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):254–280, 1984. 50

[74] Leslie Lamport. On interprocess communication. *Distributed computing*, 1(2):86–101, 1986. 47

[75] Du Li, Limin Zhou, and Richard R Muntz. A new paradigm of user intention preservation in realtime collaborative editing systems. In *Parallel and Distributed Systems, 2000. Proceedings. Seventh International Conference on*, pages 401–408. IEEE, 2000. 57

[76] Richard J Lipton and Jonathan S Sandberg. *PRAM: A scalable shared memory*. Princeton University, Department of Computer Science, 1988. 19, 24, 33, 42, 61, 62, 111

[77] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011. 65

[78] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. Technical report, University of Texas at Austin, 2011. 65

[79] George H Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955. 26

[80] Alessia Milani. *Causal consistency in static and dynamic distributed systems*. PhD thesis, PhD Thesis, "La Sapienza" Universita di Roma, 2006. 65, 117

[81] Jayadev Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(1):142–153, 1986. 103, 116

[82] David Mosberger. Memory consistency models. *ACM SIGOPS Operating Systems Review*, 27(1):18–26, 1993. 61

[83] Achour Mostefaoui, Matthieu Perrin, and Olivier Ruas. CODS D library. https://github.com/MatthieuPerrin/CODS. Accessed: 2016-06-30. 147

[84] Madhavan Mukund, Gautham Shenoy, and SP Suresh. Optimized or-sets without ordering constraints. In *Distributed Computing and Networking*, pages 227–241. Springer, 2014. 57, 60

[85] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. LSEQ: an adaptive structure for sequences in distributed collaborative editing. In *Proceedings of the 2013 ACM symposium on Document engineering*, pages 37–46. ACM, 2013. 24, 54

[86] Gil Neiger. Set-linearizability. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, page 396. ACM, 1994. 51

[87] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for P2P collaborative editing. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 259–268. ACM, 2006. 24, 53

[88] Christos H Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979. 51, 53

[89] Matthieu Perrin, Claude Jard, and Achour Mostefaoui. Construction d'une sémantique concurrente par instrumentation d'une sémantique opérationelle structurelle. In *MSR 2013-Modélisation des Systèmes Réactifs*, 2013. 159

[90] Matthieu Perrin, Claude Jard, and Achour Mostefaoui. Proof of the Instrumented Semantics for Orc. Research report, LINA-University of Nantes, January 2015. 172, 177

[91] Matthieu Perrin, Claude Jard, and Achour Mostefaoui. Tracking Causal Dependencies in Web Services Orchestrations Defined in ORC. In *NETYS-3rd International Conference on NETwork sYStems*, 2015. 159

[92] Matthieu Perrin, Achour Mostéfaoui, and Claude Jard. Brief Announcement: Update Consistency in Partitionable Systems. In *Proceedings of the 28th International Symposium on Distributed Computing*, page 546. Springer, 2014. 72

[93] Matthieu Perrin, Achour Mostéfaoui, and Claude Jard. Update Consistency for Wait-free Concurrent Objects. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015. 72

[94] Matthieu Perrin, Achour Mostéfaoui, and Claude Jard. Causal consistency: beyond memory. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 26. ACM, 2016. 105

[95] Karin Petersen, Mike J Spreitzer, Douglas B Terry, Marvin M Theimer, and Alan J Demers. Flexible update propagation for weakly consistent replication. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 288–301. ACM, 1997. 53

[96] Gordon D Plotkin. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60:3–15, 2004. 158

[97] Nuno Preguica, Joan M Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 395–403. IEEE, 2009. 24, 54

[98] Michel Raynal, André Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Information processing letters*, 39(6):343–350, 1991. 38

[99] Michel Raynal and Mukesh Singhal. Logical time: Capturing causality in distributed systems. *Computer*, 29(2):49–56, 1996. 38

[100] Sidney Rosario, Albert Benveniste, Stefan Haar, and Claude Jard. Foundations for Web Services Orchestrations: functional and QoS aspects, jointly. In *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*, pages 309–316. IEEE, 2006. 158, 176

[101] Sidney Rosario, Albert Benveniste, Stefan Haar, Claude Jard, et al. Net systems semantics of web services orchestrations modeled in Orc. Technical report, IRISA, Rennes, 2006. 176

[102] Sidney Rosario, David Kitchin, Albert Benveniste, William Cook, Stefan Haar, and Claude Jard. *Event structure semantics of orc*. Springer, 2008. 176

[103] Grigore Roşu and Koushik Sen. An instrumentation technique for online analysis of multithreaded programs. *Concurrency and Computation: Practice and Experience*, 19(3):311–325, 2007. 176

[104] Olivier Ruas, Achour Mostéfaoui, and Matthieu Perrin. Weak Consistency Criteria: Conception and Implementation. Technical report, University of Nantes, 2014. 72, 147

[105] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys (CSUR)*, 37(1):42–81, 2005. 53

[106] André Schiper, Jorge Eggli, and Alain Sandoz. A new algorithm to implement causal ordering. In *Distributed Algorithms*, pages 219–232. Springer, 1989. 38

[107] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990. 50

[108] Marco Serafini, Dan Dobre, Matthias Majuntke, Péter Bokor, and Neeraj Suri. Eventually linearizable shared objects. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 95–104. ACM, 2010. 51

[109] Jaroslav Ŝevčik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. In *ACM SIGPLAN Notices*, volume 46, pages 43–54. ACM, 2011. 15

[110] Ali Sezgin. Sequential Consistency and Concurrent Data Structures. *arXiv preprint arXiv:1506.04910*, 2015. 45

[111] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer, 2011. 55, 139

[112] Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski, et al. A comprehensive study of convergent and commutative replicated data types. Technical report, UPMC, Paris, 2011. 55, 57, 60, 139

[113] Nir Shavit. Data structures in the multicore age. *Communications of the ACM*, 54(3):76–84, 2011. 23, 180

[114] Eric Shenk. The consensus hierarchy is not robust. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, page 279. ACM, 1997. 50

[115] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011. 63

[116] UC Berkeley Space Sciences Laboratory. SETI@home. `http://setiathome.ssl.berkeley.edu/`. Accessed: 2016-06-30. 14

[117] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1):63–108, 1998. 19, 57, 65

[118] Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. Session guarantees for weakly consistent replicated data. In *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on*, pages 140–149. IEEE, 1994. 61

[119] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 172–182. ACM, 1995. 53

[120] Lewis Tseng, Alec Benzer, and Nitin Vaidya. Application-Aware Consistency: An Application to Social Network. *arXiv preprint arXiv:1502.04395*, 2015. 65

[121] Roman Vitenberg and Roy Friedman. On the locality of consistency conditions. In *Distributed Computing*, pages 92–105. Springer, 2003. 41

[122] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009. 16, 20, 49, 53, 54

[123] William E. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(2):249–282, 1989. 53

[124] Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot: a scalable optimistic replication algorithm for collaborative editing on P2P networks. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 404–412. IEEE, 2009. 24, 54

[125] Glynn Winskel. *Event structures*. Springer, 1987. 33

[126] Gene TJ Wuu and Arthur J Bernstein. Efficient solutions to the replicated log and dictionary problems. *Operating systems review*, 20(1):57–66, 1986. 57

[127] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining ACID and BASE in a distributed database. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (Berkeley, CA, USA, 2014), OSDI*, volume 14, pages 495–509, 2014. 53

Table des figures

Table des notations

Notations mathématiques

$f : \begin{cases} E \to F \\ x \mapsto f(x) \end{cases}$	Fonction f qui à $x \in E$ associe $f(x) \in F$
$\mathcal{P}(E)$	Ensembles des parties de E
$\mathcal{P}_{<\infty}(E)$	Ensembles des parties finies de E
$E \sqcup F$	Union disjointe de E et F
$f\|_E$	Restriction de f à E (f : fonction, E : ensemble)
$\lfloor e \rfloor_\prec$	Passé de e selon $\prec$ ($\prec \subset E^2$: relation acyclique, $e \in E$)
$\lceil e \rceil_\prec$	Futur de e selon $\prec$ ($\prec \subset E^2$: relation acyclique, $e \in E$)
$\prec^\star$	Fermeture transitive de $\prec$ ($\prec \subset E^2$: relation)
$\Sigma^\star$	Ensembles des mots finis sur Σ (Σ : alphabet)
Σ^ω	Ensembles des mots infinis sur Σ (Σ : alphabet)
Σ^∞	$\Sigma^\infty = \Sigma^\star \cup \Sigma^\omega$ (Σ : alphabet)
$\|u\|$	Longueur de u ($u \in \Sigma^\star$)
ε	Mot de longueur 0
$u \cdot v$	Concaténation de u et v ($u \in \Sigma^\star$, $v \in \Sigma^\infty$)

Types de données abstraits

Histoires concurrentes

Critères de cohérence

UNIVERSITE BRETAGNE LOIRE

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Matthieu PERRIN

Spécification des objets partagés dans les systèmes répartis sans-attente

Specification of shared objects in wait-free distributed systems

Résumé
Dans les systèmes répartis à très grande échelle, les critères de cohérence forts comme la cohérence séquentielle et la linéarisabilité sont souvent trop coûteux, voire impossibles à obtenir. Dans cette thèse, nous nous posons la question de la spécification des objets que l'on peut tout de même obtenir. Nous soutenons qu'il est toujours possible de séparer leur spécification en deux facettes : un type de données abstrait qui spécifie l'aspect fonctionnel des opérations et un critère de cohérence faible qui décrit la qualité de service garantie par l'objet dans son environnement réparti. Nous illustrons ces concepts par une mise en œuvre dans le langage D : les types de données abstraits sont les classes du programme et les critères de cohérence sont choisis dans une liste fournie par la bibliothèque CODS.
Nous dressons une carte de l'espace des critères faibles organisée autour de trois familles de critères primaires (localité d'état, convergence et validité) et trois familles de critères secondaires (cohérence d'écritures, cohérence pipeline et sérialisabilité). Chaque critère secondaire renforce deux critères primaires, mais les trois critères primaires ne peuvent pas être implémentés ensembles dans les systèmes considérés. Nous étudions également l'effet de la causalité sur ces familles.

Mots clés
CODS, Cohérence causale, Cohérence d'écritures, Cohérence faible, Cohérence pipeline, Cohérence séquentielle, Convergence, Critère de cohérence, Localité d'états, Orc, Sémantique opérationnelle structurelle, Sérialisabilité, Systèmes sans-attente, Type de données abstrait, Validité.

Abstract
In large scale distributed systems, strong consistency criteria like sequential consistency and linearizability are often very expensive or even unachievable. This thesis investigates the best ways to specify the objects that are still possible to implement in these systems. We assert that it is still possible to separate their specification in two complementary facets: an abstract data type that specifies the functional aspect of the operations and a weak consistency criterion that describes the level of quality of service ensured by the object in its distributed environment. We illustrate these concepts by an implementation in the D programming language: abstract data types are described by classes in the program and consistency criteria are taken from a list in the CODS library.
We also draw up a map of the space of weak consistency criteria, organised around three families of primary criteria (state locality, eventual consistency and validity) and three families of secondary criteria (update consistency, pipelined consistency and serializability). Each secondary criterion strenghtens two primary criteria, but the three criteria can not be implemented together in considered systems. We also study the effects of causality on these families.

Key Words
Abstract data types, Causal consistency, CODS, Consistency criterion, Eventual consistency, Orc, Pipeline consistency, Sequential consistency, Update consistency, Serializability, State locality, Structural operational semantics, Validity, Wait-free systems, Weak consistency.