HAL
open science

Étude de la complexité des implémentations d'objets concurrents, sans attente, abandonnables et/ou solo-rapides

Claire Capdevielle

To cite this version:

Claire Capdevielle. Étude de la complexité des implémentations d'objets concurrents, sans attente, abandonnables et/ou solo-rapides. Autre [cs.OH]. Université de Bordeaux, 2016. Français. NNT : 2016BORD0194 . tel-01414133

HAL Id: tel-01414133

https://theses.hal.science/tel-01414133v1

Submitted on 12 Dec 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Claire Capdevielle**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Étude de la complexité des implémentations d'objets concurrents, sans attente, abandonnables et/ou solo-rapides

---

Sous la direction de : **Colette Johnen et Alessia Milani**

Date de soutenance : 3 novembre 2016

Devant la commission d'examen composée de :

Rapporteurs :		
Carole Delporte-Gallet	Professeur des universités	IRIF – Université Paris Diderot
Achour Mostefaoui	Professeur des universités	Université de Nantes
Examinateur :		
Matthieu Roy	Chargé de recherche	LAAS-CNRS Toulouse
Président du jury :		
David James Sherman ...	Directeur de recherche ...	INRIA Bordeaux

– 2016 –

Titre Étude de la complexité des implémentations d'objets concurrents, sans attente, abandonnables et/ou solo-rapides

Résumé Dans un ordinateur multiprocesseur, lors de l'accès à la mémoire partagée, il faut synchroniser les entités de calcul (processus). Cela peut se faire à l'aide de verrous, mais des problèmes se posent (par exemple interblocages, mauvaise tolérance aux pannes). On s'est intéressé à l'implémentation d'abstractions (consensus et construction universelle) qui peuvent faciliter la programmation concurrente sans attente, sans utiliser de verrous mais basés sur des lectures/écritures atomiques (LEA).

L'usage exclusive des LEA ne permet pas de réaliser un consensus sans attente. Néanmoins, autoriser l'usage de primitives offrant une puissance de synchronisation plus forte que des LEA, mais coûteuse en temps de calcul, le permet. Nous nous sommes donc intéressés dans cette thèse à des programmes qui limitent l'usage de ces primitives aux seules situations où les processus sont en concurrence, ces programmes sont dit solo-rapides. Une autre piste étudiée est de permettre à l'objet, lorsqu'il y a de la concurrence, de retourner une réponse spéciale "abandon" qui signifie l'abandon des calculs en cours. Ces objets sont dit abandonnables.

D'une part, nous donnons des implémentations d'objets concurrents sans attente, abandonnables et/ou solo-rapides. Pour cela, nous proposons une construction universelle qui assure à l'objet implémenté d'être abandonnable et solo-rapide ; nous avons réalisés des algorithmes de consensus solo-rapides et des algorithmes de consensus abandonnable. D'autre part nous étudions la complexité en espace de ces implémentations en proposant des bornes inférieures sur l'implémentation des objets abandonnables et sur le consensus.

Mots-clés mémoire partagée, algorithme distribué, complexité en espace, objet concurrent, objet sans attente, objet abandonnable, implémentation solo-rapide, consensus, construction universelle, value-splitter, borne inférieure en espace

Laboratoire d'accueil Laboratoire Bordelais de Recherche en Informatique (LaBRI), Unité Mixte de Recherche CNRS (UMR 5800), 351, cours de la Libération F-33405 Talence cedex

Title On the complexity of wait-free, abortable and/or solo-fast concurrent object implementations

Abstract In multiprocessor computer, synchronizations between processes are needed for the access to the shared memory. Usually this is done by using locks, but there are some issues as deadlocks or lack of fault-tolerance. We are interested in implementing abstractions (as consensus or universal construction) which ease the programming of wait-free concurrent objects, without using lock but based on atomic Read/Write operations (ARW).

Only using the ARW does not permit to implement wait-free consensus. The use of primitives which offer a higher power of synchronization than the ARW is needed. But these primitives are more expensive in computing time. Therefore, we are interested in this thesis in the design of algorithms which restrict the use of these primitives only to the cases where processes are in contention. These algorithms are said solo-fast. Another direction is to allow the object to abort the computation in progress - and to return a special response "abort" - when there is contention. These objects are named abortable.

On the one hand we give wait-free, abortable and/or solo-fast concurrent object implementations. Indeed we proposed a universal construction which ensure to the implemented object to be abortable and solo-fast. We have also realized solo-fast consensus algorithms and abortable consensus algorithms. On the other hand, we study the space complexity of these implementations : we prove space lower bound on the implementation of abortable object and consensus.

Keywords shared memory, distributed algorithm, space complexity, concurrent object, wait-free object, abortable object, solo-fast implementation, consensus, universal construction, value-splitter, space lower bound

Remerciements

Tout d'abord je souhaite remercier les membres du jury : David Sherman pour avoir accepté de présider le jury de cette thèse ; Carole Delporte-Gallet et Achour Mostefaoui pour la relecture attentive de ce manuscrit ; ainsi que Matthieu Roy pour avoir accepté de participer à la soutenance.

Un grand merci ensuite à mes deux directrices de thèse, Colette Johnen et Alessia Milani. Ce fût grâce à elles trois années de travail passionnantes, enrichissantes et également très amicales.

Merci à Lorijn, Nesrine, Omessaad, Olfa, Thao, Thanh et Wafa pour tous ces déjeuners pris dans la bonne humeur et le partage. Merci à David, Jérôme, Joris, Noël, Romaric et Vincent pour la bonne ambiance dans le bureau et au sein de l'association AFODIB. Merci à vous toutes et tous pour les discussions, controverses et échanges qui ont animé ces trois années. De manière plus générale merci à toutes les personnes rencontrées au LaBRI qui font de ce laboratoire un endroit où il est agréable de travailler et d'apprendre.

Je souhaite remercier également ma famille et mes amis pour le soutien qu'ils m'ont apporté et d'être venu nombreux le jour de ma soutenance. Merci en particulier à ma mère pour la préparation du pot. Merci enfin à Quentin, que je remercie également pour la préparation du pot mais surtout pour sa présence et ses conseils tout au long de cette thèse.

Table des matières

Table des matières **vii**

Introduction **1**

1 Cadre de l'étude **9**
1.1 Modèle et définitions 9
1.1.1 Les objets 10
1.1.2 Pas de calcul et exécutions 11
1.1.3 Propriétés des algorithmes 12
1.1.4 Calcul de la complexité 14
1.2 État de l'art 14
1.2.1 Objets abandonnables 15
1.2.2 Implémenter des objets déterministiquement abandonnables 17
1.2.3 Complexité spatiale du k-accord et du consensus 18
1.3 Résumé des résultats et organisation du document 20

2 Construction universelle pour des objets déterministiquement abandonnables **23**
2.1 Borne inférieure 23
2.2 Une Construction universelle non-triviale et intervalle-solo-rapide non trivialement d'objets DA 27
2.2.1 Présentation de l'algorithme 27
2.2.2 Preuves 32
2.3 Résumé 43

3 Abstractions de base pour résoudre le consensus et le k-accord **45**
3.1 k-value-splitter 45
3.1.1 k-value-splitter invariant-aux-entrées 46
3.1.2 k-value-splitter non-invariant-aux-entrées 49
3.2 Étude comparée 51
3.2.1 1-value-splitter versus Splitter 51
3.2.2 1-value-splitter versus conflict-detector et adopt-commit 53
3.2.3 1-value-splitter versus grafarius 59

3.2.4 k-value-splitter versus k-converge 63
3.3 Résumé 68

4 Consensus et k-accord **69**
4.1 k-accord intervalle-solo-rapide 69
4.1.1 Borne inférieure 70
4.1.2 k-accord intervalle-solo-rapide optimal 77
4.2 Consensus avec la propriété d'abandon 81
4.2.1 Abandon déterministe 81
4.2.2 Abandon non-déterministe 84
4.3 Résumé 92

Conclusion **95**

Bibliographie **99**

Introduction

Une fable

C'est l'histoire d'un comptable à qui on fait faire des calculs. Il a son bureau sur lequel il y a sa calculatrice et des feuilles de papier sur lesquelles il écrit ses résultats. Chaque jour, son programme de la journée est apporté par le programmeur et il en suit les instructions. De temps en temps, il stocke certains de ses résultats, comme demandé par les instructions, dans la bibliothèque qui se trouve au fond du couloir. Cette bibliothèque est composée de casiers, chacun des casiers contenant une feuille. Pour lire les résultats stockés dans la bibliothèque, il va devant le casier dont il veut le résultat. Il lit le premier chiffre, puis il l'écrit sur sa feuille et fait de même avec tous les autres chiffres inscrits dans le casier. Pour écrire, il va également devant le casier, lit sur sa feuille le premier chiffre à écrire, puis l'écrit sur la feuille du casier. L'ancien chiffre disparaît automatiquement, puis il passe au second et ainsi de suite jusqu'au dernier chiffre présent sur sa feuille.

Chaque année notre comptable part en formation, ce qui lui permet d'améliorer sa vitesse de frappe sur sa calculette et ainsi d'augmenter sa rapidité de calcul. Mais voilà qu'après toutes ces formations, ce bon comptable n'arrive plus à augmenter sa vitesse. Ses muscles chauffent tellement, qu'il pourrait les faire fondre s'il accélérait encore. Comme la politique de la maison est "On double la rapidité de calcul tous les ans", on décide de construire un bureau adjacent, dans lequel est mis un autre comptable.

Le programmeur sait que les comptables sont toujours trop absorbés par leur travail pour pouvoir communiquer entre eux autrement qu'en écrivant ou en lisant dans la bibliothèque qui sera maintenant partagée.

Où l'on comprend qu'il faut communiquer et que l'utilisation de verrous peut être à double tranchant

Premier jour de travail : tout va bien. En effet, les deux comptables travaillent de leur côté et écrivent dans des casiers différents. Chacun a en effet reçu une tâche totalement différente et il n'y a donc aucun besoin de communication.

Le lendemain c'est un jour d'inventaire, la valeur de tous les casiers numérotés de 1 à 1 million doit être vérifiée. Aucun de ces casiers ne doit contenir la valeur 0. Au départ le programmeur avait pensé donner à vérifier les casiers de numéro pair au comptable expérimenté et ceux de numéro impair au jeune comptable. Cependant après réflexion, il se dit qu'il y a de fortes chances que le comptable plus ancien, de par son expérience, finisse avant le jeune comptable et lui reproche de pas avoir mieux partagé la tâche à accomplir. Ou pire encore, que ce soit le jeune qui finisse avant le plus ancien et que l'on entende ce dernier râler pendant des jours, essayant de se trouver une excuse pour sa lenteur ! Aucun comptable ne doit finir avant l'autre. Il se décide donc à faire communiquer les deux comptables en utilisant un casier pour qu'ils puissent se partager la tâche au fur et à mesure. Le casier partagé servira à définir le prochain casier à vérifier et sera incrémenté par les comptables. Comme c'est la première fois qu'il essaye de faire communiquer les comptables entre eux, il demande à un inspecteur des travaux finis de vérifier que tout se passe bien. Cet inspecteur doit vérifier que le casier partagé est bien incrémenté de un à chaque fois et que chaque casier est vérifié par un seul comptable.

Au bout de quelque temps, l'inspecteur des travaux finis les arrêtent. Le jeune : "Que se passe-t-il ?"
L'inspecteur des travaux finis : "J'observe des variations inattendues. La valeur du casier qui a été incrémentée normalement jusqu'à cent mille est repassée à quatre cent, puis après être revenue à cent mille, elle est passée directement à cent cinquante mille."
L'ancien : "Pourtant on a suivi scrupuleusement ce qu'il nous a demandé..."
L'inspecteur des travaux finis :"Je vous regardais travailler, je peux vous aider. Par exemple, lorsque la valeur est revenu à quatre cent, c'est que l'un d'entre vous s'est reposé entre sa lecture de la valeur 399 dans le casier partagé et son écriture dans ce même casier ; pendant son temps de pause, l'autre a eu le temps de vérifier de nombreux autre casiers. Pour le passage de cent mille à cent cinquante mille, c'est encore plus drôle ! L'un avait ré-avancé jusqu'à vouloir écrire la valeur cent mille, tandis que l'autre voulait encore écrire la valeur cinquante mille. Comme vous écrivez chiffre après chiffre, personne ne voulait écrire cent cinquante mille mais ce fut quand même le cas. Cerise sur le gâteau, pendant une partie de votre temps, vous vérifiiez le même casier car vous étiez synchrone et lisiez la même valeur dans le casier partagé. "
L'ancien : "C'est une journée fichue en l'air. Ha ! De mon temps ..."
Le jeune : "Il nous faut de la sûreté dans nos calculs, c'est évident ! Que pouvons nous faire ?"
L'ancien : "Pourquoi ne pas limiter l'accès de la bibliothèque à une seule personne comme lorsque j'étais seul ? Demandons au programmeur d'installer un verrou à l'entrée de la bibliothèque. "

Résolution à la fin du second jour : il est décidé qu'un verrou sera mis sur la porte de la bibliothèque. Ainsi un comptable voulant lire ou écrire dans la

bibliothèque bloquera l'accès avant de faire ses lectures et écritures.

Le troisième jour commença et il parut sans fin pour notre vieux comptable. Presque à chaque fois qu'il voulait accéder à la bibliothèque il était bloqué par le verrou et il devait attendre. L'impression négative est telle que même si ce n'était que quelques fois cela lui parut arriver à chaque fois. Comme cette impression d'être toujours dans la file la plus lente au supermarché. Il lui est même arrivé de s'endormir ; il est donc persuadé que le jeune lui est ainsi passé devant plusieurs fois, alors qu'il attendait encore pensant que la bibliothèque était verrouillée. Bref une journée fatigante à ne rien faire, ce qui n'est pas dans l'habitude de ce comptable expérimenté.

L'ancien : "Je n'ai pas fait le tiers de ce que j'aurais pu faire si j'avais été seul"
Le jeune : "Tu exagères, grâce à moi, on peut paralléliser quand même !"
L'ancien : "Certes, mais l'avancement d'aujourd'hui n'est tout de même pas satisfaisant"
Le jeune : "Tu as raison, peut être pouvons nous paralléliser un peu plus tout en maintenant la sûreté ? Ainsi l'avancement serait meilleur".
L'ancien : "Je vais proposer au programmeur de mettre des verrous plus finement, sur chaque casier par exemple. Il n'aura plus qu'à nous dire lesquels verrouiller avant que l'on écrive ou que l'on lise dans la bibliothèque."
Le jeune : "Ça me va !"

Résolution à la fin du troisième jour : Pour que les comptables n'attendent pas alors qu'ils ne souhaitent pas utiliser les mêmes casiers, il est décidé qu'un verrou sera mis sur chaque casier.

Ainsi, le quatrième jour commença avec l'espoir d'une rapidité et une efficacité redoublée. Mais voila qu'après la pause de midi un drame a lieu. Voici notre ancien comptable qui verrouille deux casiers puis va vers un troisième déjà verrouillé et attends. De son coté, le jeune comptable attends également qu'un casier soit déverrouillé par l'autre comptable. Une heure passe, puis deux, puis l'alarme d'arrêt du travail retentit.

Les comptables se dirigent vers la salle de réunion pour recevoir leurs nouvelles instructions, comme il est d'usage dans cette situation.
L'ancien : "Tu devrais travailler plus vite, je t'attends depuis des heures !"
Le jeune : "Hé, mais c'est moi qui attends sur toi !"
Le programmeur : "Pas d'affolement, pas d'affolement, j'en prends la responsabilité"
L'ancien : "C'est encore de votre faute ... Mais peut-on vous faire confiance ?"
Le programmeur : "Oui, mais rien n'est évident"
Le jeune : "Et puis de toute façon, j'ai entendu dire que dans une autre boite un des comptables s'est cassé la jambe pendant son travail et n'a pas pu déverrouiller son casier. Tous ses collègues ont attendu deux jours qu'il vienne déverrouiller le casier. Donc même sans erreur du programmeur ..."
Le programmeur : "Peut-être devrions nous penser les choses différemment ?"

Un inconnu entre.
L'ancien : "Qui êtes vous ?"
Le bibliothécaire : "Le bibliothécaire. Je suis venu vous proposer quelque chose de différent."
Le programmeur : "Quoi donc ?"
Le bibliothécaire : "Ne plus utiliser de verrous."
Le programmeur : "C'est possible ?"
Le bibliothécaire : "C'est possible."

Où l'on essaye de comprendre ce qui est possible ou non de faire avec la nouvelle bibliothèque

Et voici le grand jour de la réouverture de la bibliothèque après la fin des travaux entrepris par le bibliothécaire.

Le bibliothécaire : "Bonjour à tous, approchez vous et laissez moi vous expliquer le nouveau fonctionnement de cette bibliothèque partagée. Vous avez toujours des casiers qui contiennent une feuille chacun, mais la lecture et l'écriture ont changé. Au lieu d'écrire chiffre par chiffre, vous écrirez avec les nouveaux stylos le nombre de manière instantané. De même sur les nouvelles feuilles vous lirez instantanément le nombre et non plus chiffre par chiffre. Vous comprenez ?"
Comptable : "Oui, mais pourquoi y a-t-il des stylos et des feuilles de couleurs différentes ?"
Le bibliothécaire : "J'y viens, j'y viens. Lorsque l'on a fait des expérimentations, on s'est aperçu que quelque chose ne fonctionnait pas comme espéré."
Le programmeur : "Comment cela ?"
Le bibliothécaire : "Eh bien voilà, tu te rappelles du grand inventaire et de ton casier qui servait de compteur pour les deux comptables. Comment ferais-tu pour que cela fonctionne, sans qu'un comptable attende l'autre en n'utilisant bien sûr uniquement l'écriture et la lecture que je t'ai présenté à l'instant."
Le programmeur : "Je ne sais pas, laisse moi un peu de temps et je te donnerai le programme."
Le bibliothécaire : "En fait non, tu ne me donneras jamais ce programme, car c'est tout simplement impossible. "
Le jeune : "Alors que faire ?"
Le bibliothécaire : "C'est pour cela qu'il y a des stylos et des feuilles de couleurs différentes. Je leur ait tous donnés un petit nom. Par exemple en doré là-bas c'est le *comparer-et-échanger*."
L'ancien : "Qu'est-ce qu'il a de spécial ?"
Le bibliothécaire : "Il te permet de lire ET d'écrire de manière instantanée. Plus précisément, sur la feuille doré tu auras deux nombres inscrits, celui que tu penses être dans le casier et celui que tu veux écrire. Si celui que tu penses être dans le casier est effectivement dans le casier, tu pourras écrire la nouvelle

valeur avec le stylo doré, sinon tu ne pourras pas écrire la nouvelle valeur et tu seras prévenu que l'écriture ne s'est pas faite. Et tout cela de manière instantanée ! ”
Le jeune : “Ouaaah”
Le bibliothécaire : “Mais attention, un grand pourvoir implique de grandes responsabilités !”
Le programmeur : “C'est dangereux ! ?”
Le bibliothécaire : “Non , en fait ça risque juste d'être plus long que les écritures simples ou que les lectures simples. Il vaut mieux éviter d'en abuser.”
Le programmeur : “Donc maintenant c'est à moi de jouer. Il va falloir être efficace : allier la sûreté et l'efficacité, tout en favorisant les écritures et lectures simples.”

Pour illustrer le propos de cette thèse

En fait, la question que se pose le programmeur est la question de départ de la thèse. Mais résumons d'abord ce que nous avons appris dans cette histoire.

Auparavant, la fréquence des processeurs augmentait chaque année ce qui entraînait automatiquement une augmentation de la rapidité des calculs. En effet, dans un même laps de temps le nouveau processeur faisait plus de calculs. L'augmentation de cette fréquence n'étant plus possible, les constructeurs se sont tournés vers une augmentation du nombre de processeurs. Malheureusement, l'augmentation du nombre de processeurs, n'entraîne pas automatiquement l'augmentation de la rapidité des calculs. Les programmes étant écrit de manière séquentielle, il faut trouver des calculs à faire en parallèle aux différents processeurs pour que ceux-ci travaillent simultanément et qu'ainsi il y ait une augmentation de la rapidité de calcul par rapport à l'utilisation d'un unique processeur.

Cela entraîne, lors de l'accès à la mémoire partagée, une nécessité de synchronisation entre les entités de calcul. Il ne faut pas que l'action d'un processeur perturbe l'action des autres processeurs. D'une part au niveau de la sûreté du calcul, c'est-à-dire qu'aucun processeur ne doit mettre la mémoire dans un état non souhaité. Mais il faut également s'assurer de l'avancement des calculs dans le système. Idéalement, aucun processeur ne doit attendre les autres, à cause du caractère asynchrone des processeurs et du fait qu'ils puissent tomber en panne. Il faut donc aussi garantir des propriétés de progrès.

La synchronisation entre processeurs peut se faire à l'aide de verrous, mais comme nous l'avons vu dans l'histoire précédente, des problèmes peuvent rapidement se poser, tels que la mauvaise tolérance aux pannes ou bien la difficulté de bien programmer en évitant les inter-blocages par exemple. Dans ce cas, la sûreté est plus facilement vérifiée car les accès se font de manière séquentielle, mais le progrès peut-être plus difficile à garantir.

Ainsi, pour gérer l'accès à la mémoire, il peut être intéressant de ne pas utiliser de verrou mais d'utiliser des lectures et des écritures atomiques, c'est-à-dire de telle manière que la lecture ou l'écriture paraisse être effectuée instantanément vis-à-vis des autres processeurs. C'est ce que propose le bibliothécaire au programmeur dans la fable.

A partir de ces éléments de bas niveau, le programmeur de notre histoire peut alors construire des briques de plus haut niveau. Ces objets permettrons par la suite de programmer l'accès à la mémoire de manière plus aisée, même pour un programmeur non spécialiste. Cependant, dans ce nouveau paradigme, l'utilisation seule des lectures et des écritures atomiques limite ce qu'il est possible de faire sans qu'il y ait d'attente entre les entités du système. Un résultat central qui met en évidence cette limitation est l'impossibilité d'obtenir un consensus sans attente, uniquement avec des lectures et des écritures atomiques. En effet, il a été démontré qu'il est impossible que, étant donné au moins deux entités du système proposant une valeur, elles se mettent d'accord sur une des valeurs proposées. Et ceci de manière à ce qu'il n'y ait aucune attente entre les entités et qu'elles accèdent à la mémoire avec seulement des lectures et des écritures atomiques. Or ce type d'accord est un objet de haut niveau, qu'il est intéressant d'avoir car on sait qu'à l'aide d'un certain nombre de cet objet, il est possible de programmer n'importe quelles autres structures de données.

Il faut donc d'une certaine manière contourner cette impossibilité. C'est la question que ce pose le programmeur à la fin de la fable et la question de départ de cette thèse. Des réponses ont déjà été proposées. L'une d'entre elles, très développée dans l'état de l'art, est de considérer une propriété de progrès moins forte. Au lieu de garantir que l'opération d'un processeur se finisse quel que soit l'action des autres processeurs, si ce processeur ne tombe pas en panne ; on garantit seulement que le processeur finisse s'il est le seul à effectuer cette opération. Si plusieurs processeurs exécutent une opération en même temps, on accepte que leur calculs puissent ne pas se terminer.

Dans cette thèse, on s'intéresse à des objets qui garde la propriété forte de progrès. Il nous faut donc affaiblir d'autres propriétés.

Une autre solution proposée est de ne pas utiliser que des lectures et des écritures atomiques, mais de considérer des accès qui, toujours de manière atomique, peuvent allier une écriture et une lecture. C'est ce que propose le bibliothécaire quand il donne la possibilité au programmeur et donc aux comptables d'utiliser des stylos et feuilles de couleurs différentes. C'est ce qui se passe dans les multiprocesseurs actuels où différents types d'accès à la mémoire sont proposés (par exemple compare-and-swap ou test-and-set). Lorsque l'on souhaite que la propriété de progrès soit sans attente, ces types d'accès offrent une puissance de synchronisation plus forte.

Cependant, ces types d'accès peuvent être plus long à effectuer que les primitives de base (lecture et écriture). Nous nous sommes donc intéressés à

des programmes qui limitent l'utilisation des accès à la mémoire plus puissants à des cas où les processeurs sont en concurrence sur la mémoire. On parle alors de propriétés de solo-rapidité.

Une troisième possibilité est de permettre à l'objet de retourner une réponse spéciale "abandon" qui signifie l'abandon des calculs en cours. Ces objets sont dit abandonnables. Dans ce cas, on limite le retour de cette réponse en l'utilisant seulement lorsqu'il y a de la concurrence entre plusieurs processeurs.

Nous nous sommes intéressés, d'une part, à proposer des implémentations d'objets concurrents sans attente, abandonnables et/ou solo-rapide. D'autre part nous nous sommes intéressés à comprendre les avantages d'une solution vis-à-vis des autres en étudiant leurs coûts, c'est-à-dire en étudiant la complexité des implémentations correspondantes.

Dans le prochain chapitre nous présentons notre modèle formellement, l'état-de l'art et nous le concluons par un plan des chapitres à venir. Les chapitres suivants sont la présentation de nos résultats. Enfin nous concluons ce manuscrit avec une ouverture dans le dernier chapitre.

Chapitre 1

Cadre de l'étude

Dans l'introduction, le problème de l'implémentation d'objets à accès concurrents a été posé de manière informelle, utilisant une analogie où les processus était personnifiés sous la forme de comptables, où la mémoire était représentée par une bibliothèque, etc. Dans ce chapitre, nous présentons de manière plus formelle le cadre de notre travail pour pouvoir détailler nos contributions de manière précise. Ce qui est difficilement possible en continuant cette analogie.

Pour cela, nous commençons par présenter le modèle à mémoire partagée dans lequel se sont fait nos travaux. Nous définissons ce qui est appelé objet et les propriétés de ces derniers. Dans la deuxième section de ce chapitre, nous présentons un certain nombre de travaux liés à nos contributions. Nous revenons sur l'impossibilité énoncée dans l'introduction sur le consensus et les propriétés proposées pour contourner cette impossibilité. Nous présentons également la nouveauté de nos résultats vis-à-vis de l'existant. Enfin, dans la dernière section, nous présentons le plan de ce manuscrit.

1.1 Modèle et définitions

Le modèle utilisé pour les travaux de cette thèse est un système asynchrone à mémoire partagée, dans lequel n processus $p_1 \dots p_n$, ayant chacun un identifiant unique, communiquent à travers des objets partagés. Les processus exécutent des algorithmes qui utilisent des objets d'un certain type, nommés objets de base, fournis par le système, pour implémenter d'autres types d'objets (nommés objets de haut niveau). Pour implémenter un objet de haut niveau à partir d'un ensemble d'objets de base, les processus suivent un *algorithme* A, associant à chaque processus p un automate déterministe A_p.

1.1.1 Les objets

Spécification séquentielle

Chaque objet, de base comme de haut niveau, a un type qui décrit comment l'objet doit se comporter quand on y accède de manière séquentielle. Un type est défini par un quadruplet (Q, O, R, Δ), où Q est un ensemble d'états, O est un ensemble d'opérations, R est un ensemble de réponses, et $\Delta \subseteq Q \times O \times Q \times R$ est une spécification séquentielle du type. Un tuple (s, op, s', res) appartenant à Δ signifie que si un objet o de type T est dans l'état s quand $op \in O$ est invoqué, alors o peut changer son état vers s' et retourner la réponse res.

Pour chaque type d'objet T, on peut définir des équivalents abandonnables. Ces équivalents permettent de rajouter dans l'ensemble de réponse du type T, une réponse particulière, l'*abandon*, notée $\perp$. On distingue deux types d'objets abandonnables, les objets déterministiquement abandonnables et ceux non déterministiquement abandonnables. Dans le cas des objets déterministiquement abandonnables, la réponse abandon est retournée seulement si l'opération n'a pas pris effet. Dans le cas des objets non déterministiquement abandonnables, la réponse abandon peut être retournée même si l'opération a pris effet.

Pour chaque type $T = (Q, O, R, \Delta)$, l'équivalent déterministiquement abandonnable de T, comme défini par Hadzilacos et Toueg [2013], est noté T^{da}. T^{da} est égal à $(Q, O, R^{da}, \Delta^{da})$ où $R^{da} = R \cup \{\perp\}$ pour $\perp \notin R$, et pour tout tuple (s, op, s', res) appartenant à Δ, la spécification séquentielle Δ^{da} contient les tuples suivants : (s, op, s', res) et $(s, op, s, \perp)$. Ces deux tuples de Δ^{da} correspondent à une opération op achevée normalement et une opération op abandonnée sans avoir pris effet.

Pour chaque type $T = (Q, O, R, \Delta)$, l'équivalent non déterministiquement abandonnable de T, comme défini par Aguilera *et al.* [2007], est noté T^a. T^a est égal à (Q, O, R^a, Δ^a) où $R^a = R \cup \{\perp\}$ pour $\perp \notin R$, et pour tout tuple (s, op, s', res) appartenant à Δ, la spécification séquentielle Δ^a contient les tuples suivants : (s, op, s', res) , $(s, op, s', \perp)$ et $(s, op, s, \perp)$. Ces trois tuples de Δ^a correspondent respectivement à une opération op achevée normalement, une opération op abandonnée mais ayant été effectuée et une opération op abandonnée sans avoir été effectuée.

Quand un objet est accédé de manière concurrente par plusieurs processus, il doit se comporter comme s'il était accéder uniquement de manière séquentielle. Cela est formalisé dans le critère de cohérence de linéarisabilité (voir le paragraphe 1.1.3 sur la propriété de sûreté).

Pour éviter les confusions entre les objets de base et les objets implémentés, le terme *opération* est réservé aux objets de haut niveau, tandis que celui de *primitive* le sera pour les opérations des objets de base. De plus, il sera dit qu'une opération est *réalisée* sur un objet de haut niveau, et qu'une primitive est *effectuée* sur un objet de base. Une primitive (resp. opération) est une primitive (resp. opération) d'écriture si son application peut changer l'état de

l'objet. Sinon elles est dite primitive (resp. opération) de lecture seule.

Les objets de base

L'ensemble des opérations O d'un objet de base est un sous-ensemble des primitives *Read*, *Write* et *Compare-And-Swap* (CAS). La primitive *Read* retourne l'état de l'objet sans le changer. La primitive *Write(s')* change l'état de l'objet à s' et ne retourne rien. La primitive *CAS(s,s')* change l'état de l'objet à s' si l'état actuel est s et retourne la réponse *vrai* ; si l'état de l'objet est différent de s alors l'état ne change pas et la réponse $faux$ est retournée. Un registre est défini comme un objet de base dont l'ensemble des opérations O contient seulement les primitives *Read* et *Write*.

Pour tout objet de base donné o, l'ensemble de ses primitives est dit soit *sans mémoire*, soit *avec mémoire*. Soit o un objet de base qui supporte deux primitives f et f'. Comme défini par Fich *et al.* [1998], f écrase f' sur o, si effectuer f' et ensuite f résulte au même état qu'effectuer seulement f, en utilisant dans les deux cas les mêmes valeurs d'entrée (si existantes). Un ensemble de primitives est appelé *sans mémoire*, si toutes les primitives de l'ensemble pouvant changer l'état de l'objet écrasent les autres et s'écrasent elles-mêmes.

1.1.2 Pas de calcul et exécutions

Chaque processus possède un état local qui consiste en l'ensemble des valeurs stockées dans ses variables locales et un compteur de programme. Un calcul dans le système consiste en *pas* d'un algorithme exécuté par les processus. Chaque pas est soit :

1 l'invocation d'une opération de haut niveau ;

2 l'application d'une primitive sur un objet de base qui retourne faisant changer l'état du processus et possiblement des calculs locaux ;

3 la fin d'une opération de haut niveau avec le retour de la réponse.

Une *configuration* spécifie l'état de chaque objet de base et l'état local de chaque processus à un moment dans le temps. Dans une *configuration initiale*, tous les objets de base ont une valeur initiale spécifiée par l'algorithme et tous les processus ont leurs états initiaux.

Un pas e par un processus p est *activable* dans une configuration C, si p est sur le point d'effectuer e dans C.

Un processus est *actif* si une opération a été invoquée par le processus mais cette opération n'a pas encore produit la réponse correspondante ; sinon le processus est dit *inactif*. Nous supposons qu'une opération ne peut être invoquée que par un processus inactif. Une configuration est dite *quiescente* si tous les processus sont inactifs dans celle-ci.

Une *exécution* d'un algorithme est une séquence (possiblement infinie) $C_1, \phi_1, \ldots, C_i, \phi_i, \ldots$ de configurations alternant avec des pas, où chaque pas ϕ_i appliqué sur la configuration C_i a comme résultante la configuration C_{i+1}. Pour toute exécution finie α terminant avec la configuration C et pour toute exécution α' commençant à C, l'exécution $\alpha\alpha'$ est la concaténation de α et α' et α' est appelée une *extension* de α. Une exécution α est q-libre si aucun pas n'est effectué par le processus q dans α. Dans une exécution *solo*, seulement un processus fait des pas. On dit alors également que le processus s'exécute seul.

L'*intervalle d'exécution* d'une opération débute avec son invocation et termine quand la réponse est retournée. Une opération qui a été invoquée mais n'a pas encore reçu de réponse correspondante est une opération *incomplète* ; si elle est suivie d'une réponse correspondante elle est dite *complète*. Deux opérations op et op' sont *concurrentes* dans une exécution α, si elles sont toutes les deux incomplètes dans un préfixe fini de α. Cela implique que leurs intervalles se chevauchent. Une opération op *précède* une opération op' dans α si la réponse de op précède l'invocation de op' dans α.

Une opération invoquée par un processus p dans une exécution E est *sans concurrence*, si aucun autre processus que p est actif entre l'invocation et la réponse de cette opération. Une opération est en *concurrence d'intervalle* dans une exécution α si cette opération est concurrente avec une autre opération dans α. Une opération est en *concurrence de pas* dans une exécution si tous les pas de l'opération ne sont pas contigus dans l'exécution.

Dans une exécution infinie, un processus est dit *correct* s'il fait un nombre infini de pas ou est inactif à partir d'un certain point. Dans le cas contraire il est dit *fautif*.

1.1.3 Propriétés des algorithmes

Propriétés de sûreté Un algorithme implémentant un objet de type T assure la *linéarisabilité* (introduit par Herlihy et Wing [1990]) si pour toute exécution α et pour toute opération qui est complète et certaines opérations incomplètes dans α, il y a un point dans l'intervalle d'exécution de l'opération, appelé point de linéarisation, tel que la réponse retournée par l'opération dans α est la même que la réponse qui aurait été retournée si toutes ces opérations avaient été exécutées séquentiellement en accord avec le type T de l'objet, dans l'ordre déterminé par leurs points de linéarisation.

Propriétés d'avancement Un algorithme est *sans attente* (introduit implicitement par Lamport [1974], non implicitement par Herlihy [1991]) si dans toute exécution, chaque processus correct complète son opération en un nombre fini de ses propres pas. Un algorithme est *sans obstruction* (introduit par Herlihy *et al.* [2003]) si dans toute exécution, chaque processus correct s'exécutant

seul suffisamment longtemps retourne. Par abus de langage on parlera d'un objet sans obstruction ou sans attente pour parler des implémentations de cet objet qui respectent respectivement la propriété sans obstruction ou sans attente.

Propriétés sur les primitives utilisées Un algorithme sans attente est *intervalle-solo-rapide* si, en l'absence de concurrence d'intervalle, un processus effectue seulement des primitives Read et Write. Un algorithme sans attente est *step-solo-rapide* si, en l'absence de concurrence de pas, un processus effectue seulement des primitives Read et Write.

Un algorithme sans attente est dit *intervalle-solo-rapide non-trivialement* si pour toute exécution α en l'absence de concurrence d'intervalle, un processus effectue seulement des primitives $Read$ et $Write$; et si pour toute exécution α dans laquelle des opérations sont restées incomplètes à cause de processus fautifs, il est possible de construire une exécution α' à partir de α en ajoutant un pas de réponse pour chacune des opérations restées incomplètes à cause de processus fautifs, de tel sorte que le pas soit ajouté après le dernier pas du processus concerné et que dans α' en l'absence de concurrence d'intervalle, un processus effectue seulement des primitives $Read$ et $Write$. De manière informelle, la deuxième partie de la définition de l'intervalle-solo-rapidité non-triviale, assure qu'une opération restant incomplète à cause d'un processus fautif, ne cause pas l'utilisation infinie de primitives autres que le $Read$ et $Write$.

Propriétés sur l'utilisation de l'abandon Un algorithme qui implémente un objet abandonnable est *step-non-trivial* si pour toute exécution α chaque opération qui est abandonnée est en concurrence de pas avec une autre opération dans α.

Un algorithme qui implémente un objet abandonnable est *non-trivial* si pour toute exécution α chaque opération qui est abandonnée est en concurrence d'intervalle avec une autre opération dans α ; et si pour toute exécution α dans laquelle des opérations sont restées incomplètes à cause de processus fautifs, il est possible de construire une exécution α' à partir de α en ajoutant un pas de réponse pour chacune des opérations restées incomplètes à cause de processus fautifs, de tel sorte que le pas soit ajouté après le dernier pas du processus concerné et que dans α' chaque opération qui est abandonnée est en concurrence d'intervalle avec une autre opération dans α'. De manière informelle, la deuxième partie de la définition de la non-trivialité assure qu'une opération restant incomplète à cause d'un processus fautif, ne cause pas l'abandon d'une infinité d'autres opérations.

La *non-trivialité faible* ne garde que la première des propriétés de la non-trivialité, ainsi un algorithme qui implémente un objet abandonnable est non-

trivial faiblement si pour toute exécution α chaque opération qui est abandonnée est en concurrence d'intervalle avec une autre opération dans α.

Propriétés sur l'utilisation des identifiants des processus Un algorithme A est *anonyme* si A_p ne dépend pas de l'identifiant de p, c'est-à-dire que l'algorithme programme les processus identiquement.

Propriétés sur l'utilisation des valeurs d'entrée Un algorithme est *invariants-aux-entrées* si un processus accède à la même séquence d'objets de base dans toute exécution solo de l'algorithme, indépendamment de son entrée.

1.1.4 Calcul de la complexité

Dans nos travaux, plusieurs métriques de complexité sont utilisées. La *complexité spatiale* ou *complexité en espace* correspond au nombre d'objets de base ou d'un certain type d'objet de base (lorsque cela est précisé) que l'algorithme utilise. La *complexité en pas de calcul* correspond au nombre maximal de pas effectués lors de l'exécution de l'algorithme.

1.2 État de l'art

Nous savons depuis Fischer *et al.* [1985] et Loui et Abu-Amara [1987] que dans un système asynchrone, il existe des objets concurrents ne pouvant pas être implémentés sans attente avec seulement des primitives Read et Write. Plus précisément, Fischer *et al.* [1985] et Loui et Abu-Amara [1987] démontrent l'impossibilité de l'implémentation d'un *consensus* sans attente à n processus (avec $n > 1$) uniquement avec des registres. Un objet consensus possède une opération $propose(v)$, où v est une entrée prise d'un domaine V ($|V| \geq 2$). La valeur de sortie doit satisfaire les propriétés suivantes :

Accord Toutes les valeurs de sortie doivent être les mêmes.

Validité Chaque valeurs de sortie est une des valeurs d'entrée.

Or, comme cela a été démontré par Herlihy [1991], l'*objet consensus* est un objet universel : à l'aide d'objets consensus et de simples registres, il est possible d'implémenter de manière concurrente n'importe quel objet séquentiel.

Il a été également démontré par Herlihy [1991] que pour implémenter le consensus sans attente pour un nombre n quelconque de processus, l'utilisation de primitives CAS ou de primitives de même niveau dans la hiérarchie de Herlihy est nécessaire. De manière triviale (voir l'Algorithme 1), l'utilisation

d'une primitive CAS est suffisante pour implémenter un consensus sans attente pour un nombre quelconque de processus.

Variables partagées :
Objet de base R, initialement $\varnothing$

Procédure : $propose(v)$
1 $CAS(R, \varnothing, v)$;
2 **return** $Read(R)$

Algorithme 1: Consensus sans attente

Une manière de contourner cette impossibilité, sans utiliser des primitives autre que $Read$ et $Write$, est d'affaiblir la propriété de progrès sans attente et de considérer des implémentations sans obstruction. Cependant le défaut principal des objets sans obstruction est que de par la définition de sans obstruction, lorsque plusieurs processus font des pas il n'y aucune assurance sur la terminaison de l'opération. Les processus peuvent faire des pas éternellement sans donner de réponse. Nous nous sommes intéressés à des objets sans attente et nous avons considéré d'autres types d'affaiblissement.

Premièrement, nous nous sommes intéressés au cas où l'utilisation de primitives "fortes" est limitée à la présence de contention. L'idée d'optimiser les algorithmes concurrents lorsqu'il n'y a pas de concurrence a déjà été suggérée par Lamport [1987]. Mais la notion de solo-rapidité pour de la concurrence de pas a été définie dans Attiya *et al.* [2009]. Nous nous sommes également intéressés à considérer des objets pour lesquels on accepte que les opérations aient une réponse particulière "abandon" ; ces objets sont dits abandonnables.

Dans la section suivante, nous présentons l'état de l'art des travaux en relation avec nos résultats. Nous commençons par les résultats concernant les objets abandonnables en détaillant les différentes versions existantes. Enfin nous présentons les résultats concernant le consensus et sa généralisation le k-accord, notamment avec la propriété de solo-rapidité.

1.2.1 Objets abandonnables

Le premier papier à proposer l'idée d'objets partagés qui retournent une réponse spéciale "fail" (échec) en cas de concurrence, a été Attiya *et al.* [2009]. Plus précisément, ils s'intéressent à un objet qui retourne la réponse "fail" s'il y a une concurrence de pas et que l'opération n'a pas modifié l'état de l'objet implémenté. Ils prouvent qu'un consensus de ce type ne peut pas être implémenté seulement avec des primitives Read et Write. Ils définissent alors des objets ayant une réponse "pause" supplémentaire à celle de la réponse "fail". Cette réponse est retournée dans le but de redonner la main au processus, mais le processus doit réeffectuer la même opération.

D'autres variantes d'objets avec des réponses spéciales ont été proposées.

Premièrement par Aguilera *et al.* [2007], ils définissent de manière formelle

le type objet abandonnable que nous reprenons dans le modèle. Ils s'intéressent à des objets qui retournent la réponse "abandon" en cas de concurrence d'intervalle. Ils définissent pour cela la propriété de non-trivialité pour éviter des implémentations triviales si un processus tombe en panne (également repris dans le modèle). Ils privilégient la concurrence d'intervalle pour permettre la préservation des propriétés dans le cas de la composition d'objets. En effet, ils démontrent que les objets abandonnables step-non-trivial ne permettent pas toujours de préserver cette propriété lors d'une composition d'objets. Ils définissent également des objets abandonnables questionnables, dans lesquels une opération "QUERY" est ajoutée à la spécification séquentielle. Cette opération permet de questionner quand à la réussite de la dernière opération non "QUERY" du processus, elle peut également retourner la réponse "abandon".

Enfin, l'objet abandonnable de manière déterministe a été défini dans Hadzilacos et Toueg [2013]. Comme les précédents, ils s'intéressent à des objets qui retournent la réponse "abandon" en cas de concurrence d'intervalle et donc reprennent la propriété de non-trivialité. De manière analogue à la hiérarchie proposée par Herlihy sur le consensus, ils étudient la hiérarchie des objets déterministiquement abandonnables et la compare à celle de Herlihy. Ils démontrent ainsi qu'une telle hiérarchie existe également sur les objets déterministiquement abandonnables. Ils démontrent, notamment, qu'un objet au niveau n de la hiérarchie de Herlihy ne permet pas d'implémenter un objet au niveau $n+1$ de la hiérarchie des objets déterministiquement abandonnables. Un objet au niveau n de la hiérarchie de Herlihy est tel que l'utilisation d'un certain nombre de cet objet et des registres permettent d'implémenter un consensus sans attente pour n processus. De manière similaire, un objet au niveau $n+1$ de la hiérarchie des objets déterministiquement abandonnables est tel que l'utilisation d'un certain nombre de ce type d'objet et des registres permet d'implémenter l'équivalent déterministiquement abandonnable du consensus pour $n+1$ processus.

Le premier objet proposé par Attiya *et al.* [2009] (avec uniquement la réponse "fail") correspond à l'objet déterministiquement abandonnable de Hadzilacos et Toueg [2013] ; mais alors que les premiers considèrent la concurrence de pas, les seconds considèrent la concurrence d'intervalle. Le second type d'objets proposé par Attiya *et al.* [2009] (avec les retours "fail" et "pause") est plus proche des objets de Aguilera *et al.* [2007] car il n'assure pas, dans le cas de la réponse "pause" que l'opération ait pris effet ; Cependant alors que la réponse "pause" suppose de réeffectuer la même opération, la définition des objets abandonnables de Aguilera *et al.* [2007] n'impose pas ce comportement. De plus le papier de Attiya *et al.* [2009] se concentre sur la concurrence de pas tandis que celui de Aguilera *et al.* [2007] sur la concurrence d'intervalle. Ainsi l'utilisation des réponses particulières n'est pas permise de la même façon. Enfin la différence entre le type abandonnable défini par Aguilera *et al.* [2007] et Hadzilacos et Toueg [2013] est l'aspect déterministe du retour de la réponse

Papier	Attiya *et al.* [2009]		Aguilera *et al.* [2007]		Hadzilacos et Toueg [2013]
Réponse(s) supplémentaire(s)	fail	fail et pause	abort	abort	abort
Particularité	–	si pause, réeffectue l'opération	–	opération QUERY supplémentaire	–
Concurrence considérée	pas (step)	pas (step)	intervalle	intervalle	intervalle
Déterminisme de l'abandon	oui	oui/non	non	non	oui
Implémentation avec que des registres	non	oui	oui	oui	non

FIGURE 1.1 – Tableau résumant les variantes des objets abandonnables

"abandon". Dans le premier cas, l'opération a pu ou pas avoir lieu alors que dans le second, s'il y a un retour "abandon", l'opération n'a pas pris effet. Cela ce répercute sur la puissance des objets de telle sorte qu'il faille seulement des primitives Read et Write pour implémenter des objets non déterministiquement abandonnables et sans attente, tandis que des primitives plus fortes sont nécessaires pour certains objets déterministiquement abandonnables (Hadzilacos et Toueg [2013]). À noter que l'affaiblissement de la propriété de non trivialité permet l'implémentation d'un consensus déterministiquement abandonnable avec seulement des primitives Read et Write. Nous le prouvons en donnant une implémentation non anonyme d'un tel consensus.

Un résumé des objets proposés dans les papiers de Attiya *et al.* [2009], Aguilera *et al.* [2007] et Hadzilacos et Toueg [2013] se trouve dans le tableau de la Figure 1.1.

1.2.2 Implémenter des objets déterministiquement abandonnables

Proposée par Herlihy [1991] pour prouver l'universalité du consensus, une construction universelle est une méthode qui permet de transformer un objet séquentiel en objet concurrent linéarisable. Elle prend en entrée le code séquentiel de l'opération ainsi que ses arguments. L'intérêt d'une construction universelle est de cacher la difficulté de la programmation concurrente sans verrous. En effet, si un programmeur souhaite implémenter un objet (par exemple une file ou une pile) de manière concurrente, il le programme habituellement (*i.e.* de manière séquentielle) et appelle la construction universelle

pour exécuter son objet dans un système concurrent.

Par leur définition les objets déterministiquement abandonnables sont les plus intéressant des objets abandonnables car le programmeur sait dans le cas d'une réponse "abandon" que l'opération n'a pas été réalisée. Il peut donc choisir, en connaissance de cause, de la recommencer ou non. Cependant Hadzilacos et Toueg [2013] ont prouvé la nécessité pour l'implémentation d'un consensus pour un n quelconque de processus, d'utiliser des primitives de niveau n de la hiérarchie de Herlihy. Et comme la construction universelle est un mécanisme générique, il est nécessaire d'utiliser des primitives de niveau infini (pour nous ce sera le CAS). Nous avons donc voulu combiner dans ce mécanisme de construction universelle la propriété d'abandon déterministe non-trivial avec celle d'intervalle-solo-rapidité non triviale.

Une construction universelle pour des objets abandonnables de manière déterministe est donné dans le papier de Hadzilacos et Toueg [2013]. Cette construction peut facilement être transformée pour être solo-rapide en utilisant le consensus solo-rapide de Attiya *et al.* [2009], cependant la complexité en espace est non bornée car la construction stocke toutes les opérations effectuées sur l'objet. De plus, les opérations qui ne font qu'une lecture de l'état de l'objet modifient la représentation de l'objet implémenté et peuvent être abandonnées.

Diverses constructions universelles ont été proposées pour des objets sans attente à accès concurrent. Un bon résumé peut être trouvé dans Chuong *et al.* [2010]. Ces constructions peuvent être rendues solo-rapide en remplaçant les primitives de synchronisation forte qu'elles utilisent par leur homologue solo-rapide. Cependant, d'après nos connaissances, aucune implémentation solo-rapide d'un CAS (ou LL/SC) n'existe. Luchangco *et al.* [2003] présentent un fast-CAS dont l'implémentation essaye d'assurer qu'aucune primitive autre que $Read$ ou $Write$ n'est utilisée dans une exécution sans contention. Mais, en cas de concurrence, des opérations peuvent laisser le système dans un état qui va entraîner l'utilisation de primitives fortes alors même qu'un processus peut être seul. Ainsi leur implémentation n'est pas solo-rapide. De plus, même en utilisant le consensus solo-rapide proposé par Attiya et al. qui a une complexité spatiale de $O(n)$, il n'est pas évident de modifier les constructions universelles existantes pour qu'elles assurent les bonnes propriétés de celle que nous proposons et qui est présentée au Chapitre 2.

1.2.3 Complexité spatiale du k-accord et du consensus

L'*accord ensembliste* ou *k-accord* introduit par Chaudhuri [1993] est une généralisation de l'objet consensus. Il possède également une opération $propose(v)$, où v est une entrée prise d'un domaine V ($|V| \geq 2$). La propriété de validité reste identique à celle du consensus, tandis que la propriété d'accord est dépendante de k. Elle se transforme en :

k-Accord : Au plus k différentes valeurs de sortie sont retournées.

Borowsky et Gafni [1993]; Herlihy et Shavit [1999]; Saks et Zaharoglou [2000] démontrent que le k-accord ne peut pas être résolu en utilisant seulement des primitives Read et Write à partir du moment ou plus de k processus peuvent tomber en panne. L'universalité a été également généralisée au k-accord par Gafni et Guerraoui [2011].

Comme précédemment écrit, pour dépasser l'impossibilité, tout en se restreignant à des implémentations qui utilisent seulement des primitives moins coûteuses, *Read* et *Write*, d'autres travaux se sont intéressés à l'implémentation sans obstruction du consensus. On peut noter les travaux de Fich *et al.* [1998] donnant une borne inférieure de $\Omega(\sqrt{(n)})$ registres nécessaires, qui ont été améliorés récemment et après une décennie par Gelashvili [2015] (dans le cas anonyme), puis par Zhu [2016] (dans le cas non anonyme) donnant une borne de $\Omega(n)$. Les travaux de Aspnes et Herlihy [1990] et Guerraoui et Ruppert [2005] donnent quand à eux une borne supérieure de $O(n)$ pour l'implémentation du consensus sans obstruction. La complexité spatiale du consensus sans obstruction est donc en $\theta(n)$.

Les résultats concernant la complexité spatiale de la généralisation du consensus, *i.e.* le k-accord, ne sont pas aussi fermés. Les travaux de Delporte-Gallet *et al.* [2015] donnent une borne inférieure de $\Omega(\sqrt{n/k-2})$ pour un k-accord sans obstruction, tandis que Bouzid *et al.* [2015] donnent une borne supérieure de $O(n-k+1)$ pour ce même objet.

Quelques résultats pour les objets solo-rapides ont été donnés. Notamment, dans le cas d'un consensus intervalle-solo-rapide, Luchangco *et al.* [2003] ont donné une implémentation de complexité constante en pas de calcul et en espace. Nous nous sommes donc concentré sur l'étude d'algorithmes anonymes du k-accord intervalle-solo-rapide. Outre la curiosité intellectuelle, des raisons pratiques à l'étude d'algorithmes anonymes dans le modèle à mémoire partagée sont discutés dans Guerraoui et Ruppert [2007]. La borne inférieure que nous prouvons de $\Omega(\min(\sqrt{n/k}, \log(m/k)/\log\log(m/k)))$ pour l'implémentation anonyme pour n processus d'un k-accord intervalle-solo-rapide avec comme ensemble d'entrée V, tel que $|V| = m$ $(m \geq k+1)$, est valable également pour les k-accords sans obstruction ou abandonnables. De plus, cette borne peut être atteinte dans le cas du k-accord intervalle-solo-rapide.

On peut noter que Fich *et al.* [1998] donne déjà une borne de $\Omega(\sqrt{(n)})$. Cependant, leur borne ne s'applique ni à des objets abandonnables, ni à des objets solo-rapides. En effet, d'une part ils considèrent des algorithmes n'utilisant que des primitives *Read* et *Write* même en cas de contention et l'exécution qui est construite pour prouver la borne n'est pas sans contention. D'autre part, ils considèrent une propriété de progrès, qui assure que quelque soit la configuration pour tout processus p, il existe une exécution solo de p dans laquelle il termine. Or, comme l'exécution qui est construite n'est pas sans contention, un objet abandonnable peut tout simplement renvoyer la réponse *abandon*.

Néanmoins, pour certains consensus abandonnables, nous utilisons les bornes

du consensus sans obstruction. Pour cela nous donnons un mécanisme de transformation permettant de passer d'un consensus sans obstruction vers un consensus abandonnable et inversement, ce qui nous permet d'utiliser les bornes du consensus sans obstruction. Nous arrivons a passer outre ces bornes sur un consensus non déterministiquement abandonnable, seulement en affaiblissant la propriété de non-trivialité et dans le cas non anonyme. Nous présentons un algorithme en $O(\sqrt{(}n))$ dans ce cas.

1.3 Résumé des résultats et organisation du document

Nos contributions sont découpées en trois chapitres.

Chapitre 2 :

— Nous présentons une borne inférieure linéaire au nombre de processus sur la complexité en espace de l'implémentation d'un objet déterministiquement abandonnable intervalle-solo-rapide qui assure la propriété de non-trivialité.

— Nous présentons ensuite une construction universelle intervalle-solo-rapide pour des objets abandonnables de manière déterministe avec une propriété de non-trivialité, appelée NSUC.

Chapitre 3 :

— Nous présentons une nouvelle abstraction utile pour implémenter un objet consensus ou k-accord : l'objet k-value-splitter. Nous donnons sa définition et plusieurs implémentations.

— Nous faisons une étude comparée de cet objet avec ceux déjà définis dans la littérature servant comme bloc de base pour implémenter le consensus ou le k-accord.

Chapitre 4 :

— Nous démontrons premièrement une borne inférieure de $\Omega(\min(\sqrt{n/k}, \log(m/k)/\log\log(m/k)))$, sur le nombre d'objets de base différents nécessaires au k-accord intervalle-solo-rapide, avec n le nombre de processus et m le nombre de valeurs d'entrée possibles. A noter que cette preuve est valable également pour un k-accord sans obstruction ou abandonnable.

— Nous prouvons que cette borne peut être atteinte dans le cas k-accord intervalle-solo-rapide anonyme, en donnant un algorithme utilisant l'objet k-value-splitter.

— Pour le cas du consensus déterministiquement abandonnable nous montrons qu'avec la propriété de non-trivialité faible, il est possible d'implémenter le consensus déterministiquement abandonnable en utilisant seulement des primitives Read et Write.

— Pour le cas du consensus non déterministiquement abandonnable avec

la propriété de non-trivialité, nous prouvons son équivalence en terme de complexité en espace avec le consensus sans obstruction.

— Puis nous proposons un algorithme sous-linéaire dans le cas du consensus non déterministiquement abandonnable en affaiblissant la propriété de non-trivialité à la faible non-trivialité.

Chapitre 2

Construction universelle pour des objets déterministiquement abandonnables

Considérer des objets abandonnables et assurer la propriété d'intervalle-solo-rapidité permet d'éviter l'implémentation d'objets concurrents pouvant être bloquant, tout en favorisant l'utilisation des primitives *Read* et *Write*.

Nous nous sommes donc intéressés à l'implémentation de ces objets et à leur complexité spatiale. Pour cela nous nous sommes intéressés aux constructions universelles, qui sont un moyen générique d'implémenter des objets concurrents.

Dans ce chapitre nous présentons les résultats concernant la construction universelle intervalle-solo-rapide non trivialement et non-trivial d'objets déterministiquement abandonnables (DA). Premièrement, nous prouvons une borne inférieure sur la complexité en espace du nombre de registres requis pour une telle construction universelle. Puis, nous présentons un algorithme asymptotiquement optimal en nombre de registres, ainsi que les preuves s'y rapportant.

Ces résultats ont été publiés dans Capdevielle *et al.* [2014].

2.1 Borne inférieure

Notre démonstration de la borne inférieure sur le nombre nécessaire de registres pour une construction universelle intervalle-solo-rapide non trivialement et non-trivial d'objets déterministiquement abandonnables (DA) se base sur une preuve proposée par Attiya *et al.* [2009].

Pour démontrer une complexité en espace sur les objets solo-rapide, ils s'intéressent à un sous-ensemble non vide d'objets appelés *perturbables*, précédemment définis par Jayanti *et al.* [2000]. Ils en déduisent pour une implémentation de ce type d'objets pour n processus une complexité en espace d'au

moins $n-1$ registres.

Cette preuve ne peut pas s'appliquer directement dans notre contexte. En effet, alors que dans le papier de Jayanti *et al.* [2000] la contention est définie par l'entrelacement de pas de calcul entre deux opérations effectuées par des processus différents (step-solo-rapidité), nous considérons la contention comme la présence de deux opérations concurrentes (intervalle-solo-rapidité). Notre définition de solo-rapidité est donc plus faible que celle proposée par Attiya *et al.* [2009].

Pour notre preuve, nous utilisons une définition d'objet perturbable adaptée aux objets DA. De manière informelle, un objet est perturbable si un processus p n'ayant fait encore aucun pas, va retourner des réponses différentes à une instance d'opération selon qu'elle soit effectuée après une première exécution ou après une seconde exécution différente de la première seulement par l'ajout (d'une certaine manière) de pas d'un processus $q \neq p$. La définition formelle est la suivante :

Définition 2.1. Un objet déterministiquement abandonnable O est perturbable pour n processus, si pour toute implémentation linéarisable et non-triviale de O il y a une instance d'opération op_n par un processus p_n, tel que pour toute exécution p_n-libre (*i.e.* sans pas du processus p_n) $\alpha\lambda$ dans laquelle un processus $p_l \neq p_n$ n'effectue aucun pas dans λ et aucun processus ne fait plus d'un pas dans λ, il y a une extension de α, appelée γ, fait de pas de p_l, tel que la première réponse $res \neq \perp$ que p_n retourne lors de l'exécution solo à plusieurs reprise de op_n après $\alpha\lambda$ est différente de la première réponse $res' \neq \perp$, retournée lors de l'exécution solo à plusieurs reprise de op_n après $\alpha\gamma\lambda$.

Par une simple adaptation de la preuve du Lemme 4.7 dans Jayanti *et al.* [2000], nous prouvons que l'ensemble des objets DA qui sont perturbables est non vide. Plus précisément, nous prouvons qu'un CAS k-valué DA est perturbable. Un CAS k-valué DA est un objet DA dont l'état peut être un entier entre 1 et k et qui possède les deux primitives $Read$ et CAS. Son type est (Q, O, R, Δ), où $Q = \{1, 2, .., k\}$, $O = \{Read, CAS(u, v)$ avec $u, v \in \{1, 2, .., k\}\}$, $R = \{1, 2, .., k\} \cup \{true, false, \perp\}$ et $\forall s, u, v \in \{1, 2, .., k\}$ $\Delta = \{(s, Read, s, s)\} \cup \{(s, CAS(s, v), v, true)\} \cup \{(s, CAS(u, v), s, false)$ avec $u \neq s\} \cup \{(s, Read, s, \perp)\} \cup \{(s, CAS(u, v), s, \perp)\}$.

Lemme 2.1. *Pour tous $k \geq n$, l'objet CAS k-valué DA est perturbable pour n processus quelque soit l'état initial.*

Démonstration. On considère une implémentation linéarisable et non-triviale d'un objet CAS k-valué DA O, initialisé à n'importe quelle valeur et partagé par les processus $p_1, ..., p_n$. Soit α et λ des exécutions p_n-libre et tel que dans λ il n'y ait pas plus d'un seul pas par processus.

Soit p_l un processus qui n'effectue aucun pas dans λ. Si p_l a une opération en cours dans α, considérons γ' comme l'extension de α qui complète l'opération en cours. Soit P l'ensemble des processus qui ont une opération en cours sur O à la fin de $\alpha\gamma'$, alors $P \subseteq \{p_1, ..., p_{n-1}\} - \{p_l\}$.

Soit Q l'ensemble des processus qui commence une nouvelle opération sur O durant l'exécution de λ. Comme p_l n'effectue aucun pas dans λ et comme λ est une exécution p_n-libre, nous avons $Q \subseteq \{p_1, ..., p_{n-1}\} - \{p_l\}$. De plus, si un processus a une opération en cours dans $\alpha\gamma'$, il ne peut pas en commencer une nouvelle dans λ car aucun processus n'effectue plus d'un seul pas dans λ. Donc, nous avons $P \cap Q = \emptyset$. Et comme $P, Q \subseteq \{p_1, ..., p_{n-1}\} - \{p_l\}$, nous avons que $|P| + |Q| \leq n - 2$. Soit V l'ensemble des valeurs v tel qu'une opération $CAS(v, -)$ sur O est soit en cours dans $\alpha\gamma'$ ou commence dans λ. Par les arguments précédents, $|V| \leq n - 2$.

Soit res la valeur retournée par la première réponse qui est différente de $\perp$ après que p_n réalise une séquence d'opérations de lecture après $\alpha\lambda$. Du fait de la non-trivialité, cette valeur existe.

Soit $w \in \{1, 2, .., n\}$ tel que $w \notin V$ et $w \neq res$. Soit γ'' une extension de $\alpha\gamma'$ telle que p_l réalise une séquence d'opérations de lecture jusqu'à ce que l'une de ces opérations retourne une valeur $v' \neq \perp$, puis qu'il réalise une opération $op = CAS(v', w)$. Du fait de la non-trivialité, la séquence des lectures n'est pas infinie et l'opération $CAS(v', w)$ réussie.

Étant donné que toutes les opérations CAS en cours dans $\alpha\gamma'$ ou commencées dans λ sont de la forme $CAS(v, -)$ avec $v \neq w$, aucune d'entre elles n'est linéarisable. Donc, op est le dernier CAS réussi de $\alpha\gamma'\gamma''\lambda$.

Soit res' la valeur retournée par la première réponse qui est différente de $\perp$ après que p_n réalise une séquence d'opération de lecture après $\alpha\gamma'\gamma''\lambda$. Par la non-trivialité, cette valeur existe. Nous avons que $res' = w$ et donc par définition $res' \neq res$.

□

Pour prouver la nécessité d'avoir au moins $n - 1$ registres, nous allons prouver que lors d'une certaine exécution, du fait de la propriété de perturbabilité, l'écriture sur un ensemble de registres distincts est nécessaire et que cet ensemble peut contenir jusqu'à $n - 1$ registres. Pour cela nous utiliserons les définitions de couverture d'un objet de base et de k-couverture d'une exécution, présentées ci-dessous. Le théorème et la preuve de la borne inférieure suivent ces définitions.

Définition 2.2. Un objet de base est couvert après une exécution α si l'ensemble des primitives effectuées sur o pendant α est sans mémoire, et qu'il existe un processus dont le pas activable e, après α, est une primitive d'écriture sur o. Nous disons également que e couvre o après α.
Une exécution α est k-couverte s'il existe un ensemble de processus $\{p_{j_1}, ..., p_{j_k}\}$,

appelé *ensemble couvrant*, dont les pas activables sont des primitives d'écriture couvrant un objet de base distinct après α.

Théorème 2.2. *Soit A une implémentation non-triviale intervalle-solo-rapide non trivialement pour n-processus d'un objet perturbable déterministiquement abandonnable. La complexité en espace de A est au moins de $n-1$ registres.*

Démonstration. Pour prouver le théorème, nous construisons une exécution p_n-libre et $n-1$-couverte. La preuve s'effectue par induction. L'exécution vide est clairement une exécution 0-couverte et p_n-libre.

Supposons que α_i, pour $i < n-1$, soit une exécution i-couverte par l'ensemble couvrant $\{p_{j_1}, ..., p_{j_i}\}$ et p_n-libre. Soit λ_i le fragment d'exécution composé par les pas activables en écriture des processus $p_{j_1}...p_{j_i}$ après α_i, arrangés dans un ordre arbitraire.

Soit $p_{j_{i+1}}$ un processus qui n'appartient pas à $\{p_n, p_{j_1}, ..., p_{j_i}\}$. Comme $i < n-1$, ce processus existe. Par la non-trivialité de la propriété de solo-rapidité, le processus $p_{j_{i+1}}$ effectue des primitives sans mémoire après un nombre fini de pas lors de son exécution après α_i. Soit δ la plus petite solo exécution par $p_{j_{i+1}}$ après α_i tel que $p_{j_{i+1}}$ n'effectue plus que des primitives sans mémoire après $\alpha_i\delta$ s'il continue à s'exécuter en solo. ${\alpha_i}' = \alpha_i\delta$ est p_n-libre, de plus les pas en écriture des processus $p_{j_1}...p_{j_i}$ sont actifs à la configuration suivant immédiatement ${\alpha_i}'$.

Par définition de perturbable, il y a une extension de ${\alpha_i}'$, noté γ, par $p_{j_{i+1}}$ tel que les premières réponses (différentes de l'abandon) retournées à p_n lors de l'exécution répétée de op_n après $\alpha'_i\lambda_i$ et après $\alpha'_i\gamma\lambda_i$ sont différentes. Nous affirmons que γ contient un pas d'écriture qui se fait sur un objet de base non couvert après α'_i. Par contradiction, supposons l'inverse. Comme tous les pas dans λ_i et γ sont des primitives sans mémoire, toutes les primitives d'écriture effectuées sur un objet de base pendant γ sont écrasées par un événement de λ_i. Donc, les valeurs de tous les objets de base sont les mêmes après $\alpha'_i\lambda_i$ et après $\alpha'_i\gamma\lambda_i$. Ce qui contredit l'existence de deux réponses différentes.

Soit γ' le plus petit préfixe de γ à la fin duquel $p_{j_{i+1}}$ a un un pas d'écriture actif sur le point d'accéder à un objet de base non couvert après α'_i. Nous définissons α_{i+1} comme $\alpha'_i\gamma'$. α_{i+1} est une exécution p_n-libre et $(i+1)$-couverte. Cette dernière propriété est vraie car dans la configuration immédiatement après α_{i+1}, les processus $\{p_{j_1}, ..., p_{j_i}, p_{j_{i+1}}\}$ ont un pas d'écriture actif qui couvrent un objet de base distinct. Il en découle que A a une exécution $(n-1)$-couverte. □

Étant donné qu'une construction universelle DA permet la transformation de n'importe quel objet séquentiel en objet concurrent DA cela inclut les objets perturbables. Ainsi, s'il est possible d'implémenter une construction universelle DA avec moins de $n-1$ registres, cela est également possible pour les objets perturbables DA ce qui contredit la borne inférieure. Donc, pour implémenter

une construction universelle non-trivial solo-rapide DA, il faut au moins $n-1$ registres.

2.2 Une Construction universelle non-triviale et intervalle-solo-rapide non trivialement d'objets DA

Nous présentons dans cette section un algorithme de construction universelle non-triviale et intervalle-solo-rapide non trivialement (NSUC) d'objets DA. La complexité en espace de l'implémentation résultante de l'utilisation de cette construction universelle sur un objet perturbable est asymptotiquement optimale dans le cas où l'implémentation séquentielle de l'objet nécessite un nombre constant de registres.

2.2.1 Présentation de l'algorithme

L'algorithme NSUC utilise des registres un seul écrivain multiples lecteurs (SWMR) et des objets *Compare&Swap* (CAS) (comparer et échanger). Un objet CAS a comme ensemble d'opérations les deux primitives CAS et *Read*, précédemment définies au paragraphe 1.1.1.

Dans un premier temps, nous décrivons les variables partagées utilisées par notre construction universelle. Par la suite, nous décrivons en détail les mécanismes de l'algorithme.

Description des variables partagées

Un tableau A de n registres SWMR. Chaque registre contient un numéro de séquence. Plus précisément, un processus p_i annonce son intention de changer l'état courant de l'objet partagé en écrivant dans le registre $A[i]$ le numéro de séquence qui sera associé au nouvel état si son opération réussi. Initialement, $A[i] = 0$ pour $i = 1..n$.

Un tableau F de n registres SWMR. Le processus p_i écrit $< sv, \sigma >$ dans $F[i]$, s'il a détecté qu'il est le premier processus à annoncer son intention de définir un état pour le numéro de séquence sv. σ est un pointeur vers l'état proposé par p_i pour le numéro de séquence sv. Initialement $F[i] =< 0, \perp >$ pour $i = 1..n$.

Un tableau OS de n registres SWMR. S'il n'y a pas de contention, le processus p_i écrit les valeurs $< sv, s >$ dans $OS[i]$ où s est un pointeur vers un nouvel état de l'objet partagé calculé par p_i pendant son exécution de l'opération et sv est le numéro de séquence associé. Initialement, $OS[i] =< 0, \perp >$ pour $i = 1..n$.

Un objet CAS OC. Cet objet est utilisé en cas de contention pour décider du nouvel état de l'objet parmi ceux proposés par les opérations concurrentes. Si un processus p_i détecte de la contention, il essaye de changer la valeur de OC par le triplet $< sv, id, s >$ où id est l'identifiant du processus qui propose l'état pointé par s et associé au numéro de séquence sv. id peut être différent de i si le processus p_i a détecté qu'un autre processus p_{id} est le premier à avoir proposé un nouvel état pour sv. p_i aide alors l'autre processus à effectuer son opération. Initialement, $OC =< 0, 0, \sigma >$ où σ est le pointeur vers l'état initial de l'objet partagé.

Un tableau S de n objets CAS. Avant d'essayer de réécrire dans l'objet CAS OC, un processus écrit le numéro de séquence contenu dans OC dans S. Précisément, si la valeur de OC est $< sv, i, s >$, $S[i]$ sera mis à sv. Cela est nécessaire pour assurer qu'un processus est toujours au courant qu'une de ses opérations a réussie même si cette opération a été complétée par un autre processus. De cette manière, si $S[i] = sv$ le processus i sait que son opération qui donne l'état associé à sv a réussie. Initialement, $S[i] = 0$ pour $i = 1..n$.

Description de l'algorithme NSUC

Dans la suite, si cela n'est pas explicité, les numéros de ligne font référence à l'Algorithme 2.

À n'importe quelle configuration, le tuple avec le numéro de séquence le plus élevé stocké dans l'objet CAS OC ou dans le tableau OS contient le pointeur sur l'état courant de l'objet implémenté. Lorsqu'un processus p_i souhaite exécuter une opération o sur un objet de type T, il récupère premièrement l'état courant de l'objet, ainsi que le numéro de séquence associé. Ces informations sont stockées localement dans les variables *state* et *seq* respectivement (ligne 1). Ensuite, p_i réalise localement *op* à partir de l'état lu (ligne 2). L'algorithme NSUC suppose une fonction $APPLY_T$(*s, op, arg*) qui retourne la réponse correspondante à l'invocation de l'opération *op* sur un type T lors d'une exécution séquentielle de *op* avec comme entrée *arg* effectuée sur un état de l'objet pointé par s. $APPLY_T$(*s, op, arg*) retourne également un pointeur sur le nouvel état de l'objet.

Si *op* est une opération en lecture seul, p_i retourne immédiatement la réponse (lignes 3 à 5). Nous supposons savoir si une opération est en lecture seule. Sinon, p_i vérifie si d'autres processus exécutent concurremment une opération d'écriture. Cela se fait par la lecture des autres entrées du tableau A, en vérifiant les numéros de séquence égaux ou supérieurs à $sv + 1$.

Trois cas vont être alors distingués.

Lignes 10 à 12 Un numéro de séquence strictement supérieur à $sv + 1$ est lu. Cela implique que d'autres processus ont déjà décidé l'état pour

$sv + 1$; alors p_i abandonne.

Lignes 13 à 18 Tous les numéros de séquence lus par p_i sont strictement plus petits que $sv+1$. Alors, p_i écrit le nouvel état calculé et le numéro de séquence associé ($< sv+1, newState >$) dans le registre $F[i]$ (ligne 14). Il vérifie à nouveau la présence d'opérations concurrentes (ligne 15). Considérons à nouveau le cas où tous les numéros de séquence lus dans le tableau A par p_i sont strictement inférieurs à $sv + 1$ (l'autre cas est décrit plus bas). N'importe quel autre processus en compétition pour $sv + 1$ découvrira que p_i était le premier processus à proposer un état pour $sv+1$ et il abandonnera sa propre opération, après avoir aider p_i à compléter la sienne (lignes 21 à 24). Finalement, p_i écrit le nouvel état dans le registre $OS[i]$ et retourne la réponse de l'opération (lignes 16 à 17). L'état de l'objet associé à $sv + 1$ est celui proposé par p_i.

Lignes 19 à 35 p_i lit $sv + 1$ dans l'une des entrées du tableau. Il a donc détecté un autre processus qui essaye concurremment de décider l'état pour ce numéro de séquence. Si la détection s'est faite ligne 13, alors p_i vérifie la présence d'un processus p_j en compétition pour $sv + 1$ et qui n'a vu aucune contention (*i.e.* p_j a écrit sa proposition dans $F[j]$), ligne 14. Si ce processus existe, p_i aidera p_j à effectuer le changement de l'état de l'objet (lignes 21 à 24). En particulier, comme il y a de la contention p_i essayera d'écrire l'état calculé par p_j dans l'objet CAS OC (lignes 26 à 31). Puis il retournera abandon (lignes 32 à 34). Dans l'autre cas, p_i continue la compétition avec sa propre proposition. Il essaye d'écrire l'état qu'il propose dans OC (lignes 26 à 31) jusqu'à ce qu'une décision soit prise pour $sv + 1$. Si un processus (lui-même ou un autre l'aidant) réussi à écrire dans OC la proposition de p_i, alors p_i retourne la réponse de son opération (ligne 35). Sinon, il abandonne. Le comportement reste similaire si un processus détecte la contention ligne 15.

Fonctions utilisée par NSUC

La fonction $STATE$ retourne un pointeur sur l'état courant de l'objet partagé et son numéro de séquence.

La fonction $LEVEL_A(i)$ retourne le numéro de séquence le plus élevé lu dans le tableau A qui a été écrit par un processus autre que p_i.

Pour un numéro de séquence donné sv, la fonction $WHOS_FIRST(sv)$ retourne le couple (j, σ) où j est l'identifiant du premier processus (s'il existe) à proposer un nouvel état pour sv et σ le pointeur vers l'état proposé.

La fonction OLD_WIN essaye d'écrire seq_{OC} dans l'objet CAS $S[id_{OC}]$, si la valeur de $S[id_{OC}]$ est inférieure à seq_{OC}. Cela assure qu'un proces-

```
1  < seq, state >← STATE() ;          //Récupère l'état de l'objet
2  < newState, res >← APPLY_T(state, op, arg);
3  if op is read-only then
4  |  return res
5  end
6  seq ← seq + 1;
7  A[i] ← seq;                  //Le processus annonce son intention
8  id_new ← i;
9  seq_A ← LEVEL_A(i);
10 if seq_A > seq then              //un état a été décidé pour seq
11 |  return ⊥
12 end
13 if seq_A < seq  then                        //Le processus est seul
14 |  F[i] ←< seq, newState >;
15 |  if LEVEL_A(i) < seq  then  //Le processus est toujours seul
16 |  |  OS[i] ←< seq, newState >;
17 |  |  return res
18 |  end
19 else                          //Il y a de l'intervalle contention
20 |  < id_F, newState_F >←WHOS_FIRST(seq);
21 |  if newState_F ≠ ⊥ then     //Présence d'un premier processus
22 |  |  newState ← newState_F;
23 |  |  id_new ← id_F;
24 |  end
25 end
26 < seq_OC, id_OC, state_OC >←READ(OC);
27 while seq_OC < seq do
28 |  if (seq_OC ≠ 0) then OLD_WIN(seq_OC, id_OC);
29 |  CAS(OC, < seq_OC, id_OC, state_OC >, < seq, id_new, newState > );
30 |  < seq_OC, id_OC, state_OC >←READ(OC);
31 end
32 if (seq_OC = seq ∧ id_OC ≠ i) ∨ (seq_OC > seq ∧ READ(S[i]) ≠ seq)  then
33 |  res ← ⊥;
34 end
35 return res
```

Algorithme 2: NSUC - Code pour un processus p_i réalisant une opération *op* avec en entrée *arg* sur l'objet implémenté

sus p lent dont l'opération a réussi par la modification de l'objet CAS OC soit au courant que son opération a été réussie. En effet, il se peut qu'un processus p ne fasse pas de pas pendant qu'un autre processus complète son opération. Ensuite une autre opération réécrit les changements écrits dans OC. p peut alors quand même récupérer le statut de son opération en vérifiant la valeur de $S[p]$ et ainsi retourner la réponse correcte.

```
1 seq_max ← 0 ; σ_max ← ⊥;
2 for j = 1..n do
3 |   < seq_OS, σ > ← OS[j];
4 |   if seq_OS > seq_max then seq_max ← seq_OS ; σ_max ← σ ; end
5 end
6 < seq_OC, id_OC, σ_OC > ← READ(OC);
7 if seq_OC < seq_max then return < seq_max, σ_max > end
8 return < seq_OC, σ_OC >
```

Algorithme 3: fonction $STATE()$

```
1 seq_max ← 0;
2 for j = 1..n | j ≠ i do
3 |   seq_A ← A[j];
4 |   if seq_max < seq_A then seq_max ← seq_A ; end
5 end
6 return seq_max
```

Algorithme 4: fonction $LEVEL_A(i)$

```
1 for j = 1..n do
2 |   < seq_F, σ_F > ← F[j];
3 |   if seq_F = sv then return < j, σ_F > end
4 end
5 return < 0, ⊥ >
```

Algorithme 5: fonction $WHOS_FIRST(sv)$

Complexité

Soit t le nombre maximum de pas pour réaliser une opération sur l'implémentation séquentielle de l'objet (*i.e.* la complexité en pas de calcul de la fonc-

```
1 seq_S ← READ(S[id_OC]);
2 if seq_OC > seq_S then CAS(S[id_OC], seq_S, seq_OC) end
```

Algorithme 6: fonction $OLD_WIN((seq_{OC}, id_{OC})$

tion $APPLY_T$). Par l'inspection du pseudo-code, il est aisé de voir que la complexité en pas de calcul des fonctions $STATE$, $LEVEL_A$ et $WHOS_FIRST$ est en $O(n)$. La complexité en pas de calcul de la fonction OLD_WIN, qu'en à elle, est $O(1)$. Par corollaire du Lemme 2.6, un processus peut répéter la boucle (lignes 27 à 31 de l'Algorithme 2) au plus n fois. Donc, la complexité en pas de calcul de notre construction est en $O(n+t)$. Étant donné que l'exécution de toute opération inclus l'exécution des fonctions $STATE$ et $APPLY_T$, la complexité en pas de calcul de la construction NSUC est en $\Theta(n+t)$.

Soit s la taille de la représentation séquentielle de l'objet. L'algorithme NSUC stockes au plus $2n+1$ représentations séquentielles de l'objet (n dans le tableau F, n dans le tableau OS et 1 dans OC). La complexité en espace de algorithme NSUC est en $O(ns)$.

Variante

Il est possible de n'utiliser qu'un seul objet CAS au lieu du tableau S de n objets CAS. Cependant, cet objet doit stocker toutes les informations que stocke le tableau de n objets CAS, c'est-à-dire les n numéros de séquence des dernières opérations réussies de chaque processus. Les preuves suivantes sont données avec la version du tableau S de n objets CAS.

2.2.2 Preuves

Nous prouvons par la suite les propriétés de sans attente, d'abandon déterministe, de non-trivialité, d'intervalle-solo-rapidité non triviale et de linéarisabilité. Dans la suite, tous les numéros de lignes font référence à l'Algorithme 2, sauf lorsque cela est spécifié.

Sans attente

Pour prouver la propriété de sans attente, c'est-à-dire que tous les processus non fautifs terminent après un nombre fini de leur propre pas, il faut s'assurer qu'aucun processus ne sera bloqué dans la boucle des lignes 27 à 31. Tout le reste de l'algorithme étant trivialement sans attente. Nous démontrons cela en montrant quelles sont les fragments d'exécution qui empêchent la sortie de la boucle, puis en montrant que ces cas sont en nombre fini pour une instance d'opération donné.

Pour chaque pair de tuples t_1, t_2 stockés dans OC ou dans OS, t_1 est dit plus grand que t_2 si et seulement si la séquence dans t_1 est plus grande que la séquence dans t_2.

Observation 2.3. *Les valeurs écrites dans l'objet CAS OC sont de plus en plus grandes.*

Cela résulte du fait que l'objet CAS OC est écrit seulement à la ligne 29 et que ligne 27, la valeur écrite est supérieure à la valeur stockée dans OC immédiatement avant l'écriture réussie.

Observation 2.4. $\forall i = 1..n$ *les valeurs écrites dans* $OS[i]$ *sont de plus en plus grandes.*

$OS[i]$ est seulement écrit par le processus p_i. L'inspection du pseudo-code de la fonction $STATE$ révèle que le processus p_i commence l'exécution de l'opération op avec une valeur de séquence seq égale ou supérieure à la valeur dans $OS[i]$ à la configuration avant l'invocation de op. Avant d'écrire la nouvelle valeur de séquence dans $OS[i]$, la seule opération sur op est son incrémentation (ligne 6 de l'Algorithme 2).

Lemme 2.5. *Un processus p reste dans la boucle (ligne 27 à 31) lorsqu'il exécute la ligne 27 seulement si un autre processus q réussit le CAS ligne 29 avec un numéro de séquence inférieur à la valeur de la séquence de p, entre le moment de la dernière lecture de OC par p et la dernière opération CAS sur OC par p.*

Démonstration. Supposons qu'aucun processus ne réussisse un CAS sur OC entre la dernière lecture de OC par p et l'application du CAS sur OC par p. Alors le processus p réussit son CAS et écrit un triplet avec la séquence seq dans OC. p sort de la boucle, en effet, par l'Observation 2.3 la valeur lue successivement dans OC est supérieur ou égale à seq. De manière similaire, si un processus réussit un CAS avec une séquence supérieure ou égale à seq, par l'Observation 2.3, p sort de la boucle. □

Lemme 2.6. *Soit un processus p dans la boucle (ligne 27 à 31). Un autre processus q peut empêcher p de sortir de la boucle au plus une fois.*

Démonstration. D'après le Lemme 2.5, un processus q peut empêcher un autre processus p de sortir de la boucle seulement s'il réussit son CAS avec un numéro de séquence plus petit que la valeur contenue dans la variable seq de p, entre le moment de la dernière lecture de OC par p et le CAS effectué par p après cette lecture. Soit v la valeur de la variable seq de p lorsqu'il exécute la ligne 27. Supposons qu'un processus q écrit dans OC une valeur de séquence plus petite que v entre la dernière lecture de OC par p et l'application successive d'un CAS dans OC. Après son CAS, q sort de la boucle par l'Observation 2.3 et

son opération se termine. Si q exécute une nouvelle opération, il obtiendra (à la ligne 1) un numéro de séquence plus grand ou égal à v. Cela provient de l'Observation 2.3, l'Observation 2.4 et par le pseudo-code de la fonction $STATE$. Ainsi q ne peut plus empêcher p de sortir de la boucle. □

Le Lemme 2.6 prouve le théorème suivant.

Théorème 2.7. *Chaque invocation d'une opération par un processus non fautif retourne après un nombre fini de ses propre pas.*

Objet déterministiquement abandonnable

Dans cette partie nous prouvons que chaque opération qui est abandonnée (*i.e.* retourne $\perp$) ne modifie pas l'état de l'objet implémenté. En effet, à cause du mécanisme d'aide, une opération pourrait changer l'état de l'objet sans que le processus à l'origine de cette opération le sache, et donc ce dernier abandonnerait. Cependant, nous prouvons que la présence de la fonction OLD_WIN et la lecture d'une case du tableau S empêche ce scénario d'arriver et donc assure la propriété d'abandon déterministe.

Lemme 2.8. *Pour tout entier v, au plus un processus écrit v dans F.*

Démonstration. Un processus p_i écrit v dans F (ligne 20 de l'Algorithme 2) seulement si la valeur de seq_A lue ligne 9 est plus petite que v. Observons que p_i écrit v dans $A[i]$ (ligne 7) avant d'exécuter la ligne 9 de l'Algorithme 2.

Supposons par contradiction qu'il y a un autre processus p_j qui écrit v dans F. Alors, étant donné que p_i et p_j vérifient la condition $seq_A < v$, aucun d'eux ne lit la valeur v écrite dans A par l'autre processus. Cela veut dire que la lecture de $A[i]$ (resp. $A[j]$) par p_j (resp. p_i) précède l'écriture de $A[i]$ (resp. $A[j]$) par p_i (resp. p_j). Comme l'écriture de $A[i]$ précède la lecture de $A[j]$ par p_i et similairement pour p_j, nous aboutissons à une contradiction. □

Lemme 2.9. *Soit t, t' deux tuples écrit dans OC et/ou dans OS. Si t et t' ont le même numéro de séquence, alors l'élément correspondant à l'état dans t est égal à celui dans t'.*

Démonstration. Par l'Observation 2.3, pour tout entier v, au plus un tuple avec la valeur de séquence v est écrite dans C. D'après le Lemme 2.8, il en est de même pour les tuples écrits dans OS. Il reste donc à prouver que pour toute paire de tuple t et t' avec la même valeur de séquence et tel que t est écrit dans OC et t' dans OS, la composante état de t et celle de t' est la même.

Supposons qu'il y ait un état s associé à la valeur de séquence v qui soit écrit dans OC par le processus p_i et qu'il y ai un état s' associé également à la valeur de séquence v qui soit écrit dans OS par un processus p_j. Alors p_j a écrit précédemment $< v, s' >$ dans $F[j]$ (ligne 20) et avant cette écriture p_j a écrit dans A (ligne 7). Comme à la ligne 21 p_j lit une valeur de séquence

inférieure à v, la lecture de $A[i]$ précède l'écriture de v dans $A[i]$ par p_i. Étant donné que p_i écrit v dans $A[i]$ et lit ensuite $A[j]$, la valeur lue dans $A[j]$ est supérieure ou égale à v. De plus p_i écrit dans OC (*i.e.* il exécute la ligne 29), donc la valeur lue dans $A[j]$ est égal à v. Alors, il exécute les lignes 13 à 18 et d'après le Lemme 2.8 et le pseudo-code de la fonction $WHOS_FIRST$, nous avons qu'à la ligne 14 p_i lit l'état écrit par p_j. Donc $s = s'$.

□

Observation 2.10. *Un numéro de séquence v ne peut pas être écrit dans $S[i]$ si aucun processus n'a écrit dans OC le triplet $< v, i, state >$.*

Cela découle des lignes 26, 28 de l'Algorithme 2 et de l'Algorithme 6.

Observation 2.11. *$\forall\ i = 1..n$ les valeurs écrites dans $S[i]$ sont de plus en plus grandes.*

Cela résulte de l'Algorithme 6.

Lemme 2.12. *Considérons une invocation de $OLD_WIN(v, i)$ par un processus p qui retourne de l'opération. Si p pendant son exécution de $OLD_WIN(v, i)$ lit $S[i] < v$, alors soit p réussie le CAS $CAS(S[i], -, v)$, soit un autre processus a écrit v dans $S[i]$ à une configuration qui précède l'application de son CAS.*

Démonstration. Soit $s_1, \ldots s_h$ la séquence des tuples avec l'identifiant de processus i écrit dans OC ordonnés d'après leur numéro de séquence. D'après l'Observation 2.3, cet ordre correspond à l'ordre d'écriture de ces tuples dans OC. Donc, s_k avec $k = 1, \ldots h$ est le k-ème tuple avec l'identifiant i écrit dans OC. La preuve se fait par induction sur k.

Pour le cas de base $k = 1$. Soit C la première configuration à laquelle la valeur de OC est s_1. Soit q le processus qui écrit sur la valeur s_1 de OC. Avant d'appliquer son opération d'écriture sur OC, q invoque $OLD_WIN(s_1.seq, i)$ où $s_1.seq$ est le numéro de séquence de s_1. Comme q est le processus qui écrit sur s_1 et d'après l'Observation 2.10, la valeur lue par q dans $S[i]$ est la valeur initiale et donc plus petite que $s_1.seq$. Supposons que q échoue le $CAS(S[i], -, s_1.seq)$. Alors, par l'Observation 2.10, un autre processus a écrit $s_1.seq$ dans $S[i]$ avant que q applique son CAS. Donc, lorsque q retourne de $OLD_WIN(s_1.seq, i)$ la valeur $s_1.seq$ a été écrite dans $S[i]$.

Considérons un autre processus p qui invoque $OLD_WIN(s_1.seq, i)$ et retourne de cet appel. Supposons que p lit une valeur plus petite que $s_1.seq$ et échoue le CAS $CAS(S[i], -, s_1.seq)$. S'il échoue après que q retourne de $OLD_WIN(s_1.seq, i)$, la preuve est faite. Sinon par l'Observation 2.10, un autre processus a écrit $s_1.seq$ dans $S[i]$ entre son opération de lecture et l'application du CAS.

Pour le pas d'induction, soit p un processus qui exécute $OLD_WIN(s_k.seq, i)$ et retourne de son invocation. Supposons que p lise une valeur $S[i] < s_k.seq$ et

échoue le $CAS(S[i], -, s_k.seq)$. Pour invoquer $OLD_WIN(s_k.seq, i)$, p a précédemment lu s_k dans OC (ligne 26 de l'Algorithme 2). Donc, d'après l'Observation 2.3, le tuple s_{k-1} a été écrit dans OC avant cette opération de lecture.

Soit C la première configuration pour laquelle la valeur de OC est s_{k-1}. Le processus qui écrit sur ce tuple a précédemment invoqué et retourné de l'appel de $OLD_WIN(s_{k-1}.seq, i)$. Donc, par l'hypothèse d'induction et l'Observation 2.10 à une configuration avant l'écriture dans OC de la valeur s_k, la valeur de $S[i]$ est égale à s_{k-1}.

Si p est le processus qui écrit sur la valeur de s_k dans OC, alors d'après les Observations 2.10 et 2.11, un autre processus a écrit $s_k.seq$ dans $S[i]$ avant que p applique son CAS. Sinon p n'est pas le processus qui écrit sur la valeur s_k dans OC. Dans ce cas, s'il échoue son CAS après que s_k soit écrit dans OC, la preuve est faite. Sinon, par les Observations 2.10 et 2.11, un autre processus a écrit $s_k.seq$ dans $S[i]$ avant que p applique son CAS. □

Quand un processus p_j réussit un CAS dans OC en écrivant sur $< v, i, - >$, $S[i]$ contient v.

Lemme 2.13. *Soit C la configuration immédiatement après une application réussie de $CAS(OC, < v, i, - >, -)$. Alors $S[i] = v$ à C.*

Démonstration. D'après l'Observation 2.3, un tuple $< v, -, - >$ peut être écrit dans OC seulement une fois. Soit C la configuration immédiatement après une application réussie de $CAS(OC, < v, i, - >, < -, -, - >)$ par un processus p. Comme le CAS est réussi, il y une configuration C' qui précède C tel que la valeur de l'objet CAS OC est $< v, i, - >$ à C' et entre C' et C aucune opération CAS n'a réussi sur OC.

L'inspection du pseudo-code de l'Algorithme 2 révèle que p exécute $OLD_WIN(v, i)$ avant C. Par les Observations 2.3 et 2.11, la valeur lue par p dans $S[i]$ (Algorithme 6) est plus petite ou égale à v. Si la valeur lue est inférieure à v, alors par le Lemme 2.12, soit p réussie le CAS de l'Algorithme 6 en écrivant v dans $S[i]$, soit un autre processus le fait à une configuration qui précède l'application du CAS sur $S[i]$ par p. Comme aucune opération de CAS ne réussie entre C' et C, et par l'Observation 2.11, la valeur de $S[i]$ est v à C.

□

Soit p_i un processus exécutant une opération. Nous définissons seq^i_{op} comme la valeur de la séquence que la fonction $STATE$ retourne ligne 1 durant l'exécution de l'instance de l'opération op par le processus p_i. S'il n'y a pas d'ambiguïté ou si le processus exécutant op ne nous intéresse pas, la notation seq_{op} est utilisée. Nous notons $seq_O(C)$ la valeur de séquence la plus élevée stockée dans le registre CAS OC et dans le tableau OS à la configuration C et $seq_A(C)$ la plus grande valeur dans A à la configuration C.

Lemme 2.14. *Pour tout entier positif v, si $seq_O(C) \geq v$, alors un processus a écrit $< v, -, - >$ dans OC ou $< v, - >$ dans OS avant la configuration C.*

Démonstration. La preuve se fait par induction descendante sur v. Si $v = seq_O(C)$ le lemme est juste par définition de $seq_O(C)$. Supposons que pour un v tel que $v \leq seq_O(C)$ la proposition soit vraie et prouvons que cela est vrai pour $v-1$. Soit p un processus qui a écrit $< v, -, - >$ dans OC ou $< v, - >$ dans OS avant la configuration C. Avant son opération d'écriture p a lu une valeur $v-1$ dans OC ou dans OS lorsqu'il a exécuté la ligne 1 de l'Algorithme 2. L'inspection du pseudo-code de la fonction $STATE$ révèle qu'avant la lecture et donc avant la configuration C, un processus a écrit $< v-1, -, - >$ dans OC ou $< v-1, - >$ dans OS. □

Le théorème suivant déclare qu'une instance d'opération qui est abandonnée ne change pas l'état de l'objet partagé.

Théorème 2.15. *Si une instance d'opération op exécutée par un processus p est abandonnée, alors le tuple avec le nouvel état calculé par p pendant sont exécution de op ne sera jamais écrit, ni dans OC ni dans OS.*

Démonstration. Un processus peut abandonner aux lignes 11 et 33. Si un processus abandonne ligne 11, c'est trivial.

Si un processus p_i abandonne à la ligne 33 soit $(v_{OC} = seq^i_{op}+1 \wedge id_{OC} \neq i)$, soit $(v_{OC} > seq^i_{op}+1 \wedge READ(S[i]) \neq seq^i_{op}+1)$, avec v_{OC} et id_{OC} respectivement la valeur de la séquence et l'identifiant du dernier tuple que p_i a lu dans OC.

— dans le premier cas, p_i a lu l'état associé à sa valeur de séquence $seq^i_{op}+1$ et ce n'est pas son état. D'après le Lemme 2.9, l'état qu'il propose ne sera jamais écrit dans OC ni dans OS.
— Dans le second cas, la valeur de séquence lue par p_i dans OC est supérieure à $seq^i_{op}+1$ et $S[i]$ ne contiens pas $seq^i_{op}+1$. Alors, d'après le Lemme 2.13 et l'Observation 2.11, aucun processus n'a écrit dans OC l'état proposé par p_i pour la valeur de séquence $seq^i_{op}+1$. D'après le Lemme 2.14, un tuple avec une valeur de séquence égale à $seq^i_{op}+1$ a été écrit dans OS ou dans OC. Par conséquent, par le Lemme 2.9, l'état proposé par p_i pour la valeur de séquence $seq^i_{op}+1$ ne sera jamais écrit dans OC ni dans OS.

□

Non-trivialité

Dans cette section nous prouvons que notre algorithme est non-trivial suivant la définition proposé par Aguilera *et al.* [2007]. En particulier, une opération peut être abandonnée seulement s'il y a une autre opération concurrente. De plus, une opération qui ne finit pas peut causer l'abandon d'un nombre fini d'autres opérations. Dans notre cas, nous prouvons qu'une opération peut causer l'abandon d'au plus deux autres opérations par processus. De manière

informelle, cela provient de 3 éléments : premièrement du fait qu'une opération associée à un numéro de séquence peut causer l'abandon d'une autre opération que si cette dernière à un numéro de séquence inférieure. Deuxièmement un processus ne réalise pas deux instances d'opération avec le même numéro de séquence, celui-ci est incrémenté. Enfin une opération qui débute après une autre a un numéro de séquence supérieur ou égal à l'autre (qui a débuté avant).

Observation 2.16. *Pour toute configuration C, nous avons $seq_A(C) \geq seq_O(C)$.*

Cela provient du fait que tous les processus écrivent la plus grande valeur lue dans OC et dans OS, plus un dans A avant d'écrire dans OC ou dans OS.

Lemme 2.17. *Pour toute exécution donnée α et pour toute configuration C de α, nous avons $seq_A(C) = seq_O(C)$ ou $seq_A(C) = seq_O(C) + 1$.*

Démonstration. Fixons une exécution. Initialement l'énoncé est vrai : $seq_A(0) = 0$ et $seq_O(0) = 0$. Supposons l'énoncé vrai jusqu'à une configuration C.

Deux cas se posent :

- Le pas qui amène de la configuration C à la suivante $C + 1$ est une écriture dans OC ou dans OS d'une valeur de séquence par un processus p. Dans ce cas, les valeurs dans A ne changent pas des configurations C à $C+1$, donc $seq_A(C) = seq_A(C+1)$. De plus, d'après les Observations 2.3 et 2.4, nous avons que $seq_O(C+1) \geq seq_O(C)$. Si $seq_O(C+1) = seq_O(C)$, alors l'énoncé est trivialement vrai. Sinon $seq_O(C+1) > seq_O(C)$ et donc $seq_O(C+1) \geq seq_A(C+1)$. Comme par l'observation 2.16 $seq_A(C+1) \geq seq_O(C + 1)$, nous avons finalement $seq_A(C + 1) = seq_O(C + 1)$.
- Le pas qui amène de la configuration C à la suivante $C + 1$ est une écriture dans A d'une valeur de séquence v par un processus p_i lors de l'exécution d'une opération op. Cela implique que les valeurs dans OC ou dans OS ne changent pas des configurations C à $C + 1$, donc $seq_O(C) = seq_O(C + 1)$. De plus, $v = seq^i_{op} + 1$ et $seq_O(C) \geq seq^i_{op}$ car seq^i_{op} a été lue avant la configuration C.
  - Considérons d'abord que $seq_O(C) = seq^i_{op}$. Si la valeur écrite par p_i n'est pas plus grande que les valeurs stockées dans A à la configuration C, alors $seq_A(C) = seq_A(C+1)$. Sinon, nous avons $seq_A(C+1) = seq^i_{op} + 1 = seq_O(C + 1) + 1$. Dans les deux cas l'énoncé est prouvé.
  - Considérons maintenant que $seq_O(C) > seq^i_{op}$. Alors il y a un processus qui a écrit $seq_O(C)$ dans OC ou dans OS avant la configuration C et après la configuration qui suit immédiatement la lecture par p_i à la ligne 1. Ainsi p_j a écrit $seq_O(C)$ dans $A[j]$ avant la configuration C. De plus p_j n'est pas le processus p_i, sinon $seq_{op} = seq_O(C)$, ce qui est une contradiction. Étant donné que $i \neq j$, et comme la valeur écrite par p_i est plus petite ou égale à $seq_O(C)$, nous avons $seq_A(C) = seq_A(C + 1)$. L'énoncé est prouvé.

□

Le lemme suivant établit que, lorsqu'une opération qui essaye d'écrire le i^{me} état de l'objet est abandonnée, alors le i^{me} état est déjà défini. Donc notre algorithme assure également, que même en présence de concurrence d'intervalle, au moins une des opérations réussie à modifier l'état de l'objet.

Lemme 2.18. *Soit op une opération exécutée par le processus p qui peut changer l'état de l'objet. Soit C la configuration immédiatement après le retour de op. Un tuple correspondant à la valeur de séquence $seq_{op} + 1$ a été écrite dans OC ou dans OS avant C.*

Démonstration. Soit p un processus qui exécute une opération op qui peut changer l'état de l'objet. op retourne car p exécute l'une des lignes suivantes : ligne 11, ligne 23 ou ligne 34 de l'Algorithme 2. Soit C la configuration immédiatement avant le retour de op.

- op retourne à la ligne 23 : Avant de retourner, le processus p a écrit dans son entrée de OS un tuple $< seq_{op} + 1, - >$ (ligne 22).
- op retourne à la ligne 11 : La valeur lue par le processus p à la ligne 9 est plus grande que $seq_{op} + 1$. D'après le Lemme 2.17, $seq_A(C) = seq_O(C)$ ou $seq_A(C) = seq_O(C) + 1$, donc nous avons que $seq_O(C) \geq seq_{op} + 1$. Par le Lemme 2.14, un processus a écrit un tuple avec la valeur de séquence $seq_{op} + 1$ dans OC ou dans OS avant la configuration C.
- op retourne à la ligne 35 : A cause de la ligne 27 de l'Algorithme 2, $seq_O(C) \geq seq_{op} + 1$. Donc, d'après le Lemme 2.14, un processus a écrit un tuple avec la valeur de séquence $seq_{op} + 1$ dans OC ou dans OS avant la configuration C.

□

Lemme 2.19. *Pour toute exécution donnée α, soit p une instance d'opération exécutée par le processus p_i. Si la valeur de $LEVEL_A(i)$ lue par p_i à un moment donné de l'exécution de op est plus grande ou égale à $seq^i_{op} + 1$, alors op est en concurrence avec une autre opération op' dans α.*

Démonstration. Soit p_i un processus qui exécute une opération op. Supposons que p_i lit $LEVEL_A(i) = v_A \geq seq^i_{op} + 1$ pendant son exécution de op. Alors un autre processus p_j a écrit la valeur v_A dans $A[j]$ pendant l'exécution d'une instance d'opération op'.

Supposons par contradiction que op et op' ne soient pas concurrentes. Comme les opérations ne sont pas concurrentes et que le processus p_i a lu une valeur écrite par p_j pendant op', op' précède op. L'inspection du pseudocode révèle que op et op' ne sont pas des opérations triviales étant donné qu'il accède au tableau A. D'après le Lemme 2.18, quand op' termine à la configuration C, $seq_O(C) = v_A$. Donc, à la configuration, C' lorsque p_i débute son opération $seq_O(C') \geq v_A$ et seq^i_{op} devrait être plus grand ou égal à v_A. Ce qui est une contradiction. □

Observation 2.20. *Une instance d'opération op exécuté par un processus p_i est abandonné seulement si p_i lit $LEVEL_A(i) \geq seq^i_{op}+1$ à un moment donné de l'exécution de op.*

Cela provient du fait que si une instance d'opération est abandonnée à la ligne 11, le processus a exécuté la ligne 10 ; si une instance d'opération est abandonnée à la ligne 33, le processus a exécuté la ligne 13 ou la ligne 21.

Lemme 2.21. *Pour toute exécution donné α, soit op une instance d'opération exécutée par un processus p qui est abandonnée en α. Alors op est concurrente avec une autre opération op' en α.*

Démonstration. Soit op une instance d'opération qui est abandonnée par le processus p_i. Par l'Observation 2.20, p_i a lu à un moment donné de l'exécution de op que $LEVEL_A(i) \geq seq^i_{op} + 1$. De plus, d'après le Lemme 2.19, op est concurrente avec une autre opération op' en α. □

Lemme 2.22. *Soit une exécution donnée α. Soit p_i un processus fautif en α pendant l'exécution d'une opération op. Pour tout processus non fautif p_j, définissons op_1 comme la première exécution par p_j qui est concurrente avec op. Soient op_2 et op_3 les deux instances d'opération consécutives exécutées par p_j immédiatement après op_1. Alors, nous avons que $seq^j_{op_3} > seq^i_{op}$.*

Démonstration. Soit p_i un processus fautif en α pendant l'exécution d'une opération op. Par le Théorème 2.7, toute opération exécutée par un processus non fautif termine. Soit p_j un processus non fautif en α. Par le fait que l'invocation de op_2 par p_j suit l'invocation de op par p_i et par les Observations 2.3 et 2.4, $seq^j_{op_2} \geq seq^i_{op}$. Supposons que $seq^j_{op_2} = seq^i_{op}$ (autrement l'énoncé est prouvé). Soit C' la configuration immédiatement après le retour d'op_2, d'après le Lemme 2.18, $seq_O(C') > seq^j_{op_2}$. Ainsi, l'inspection du pseudo-code de la fonction $STATE$ révèle que $seq^j_{op_3} > seq^i_{op}$. □

Lemme 2.23. *Soit une exécution donné α. Soit p_i un processus qui est fautif en α pendant l'exécution d'une opération op. Alors, pour tout processus non fautif p_j, op peut causer l'abandon de deux opérations de p_j.*

Démonstration. Soit p_i un processus fautif et p_j un autre processus. Si $seq^i_{op} < seq^j_{op}$, le processus p_i ne peut pas provoquer l'abandon de p_j. Cela provient de l'Observation 2.20 et du fait que $seq^i_{op} < seq^j_{op}$. En fait, comme $A[i] \leq seq^i_{op}+1 < seq^j_{op}+1$, p_i ne peut pas causer la lecture de $LEVEL_A(j) \geq seq^j_{op}+1$ par p_j. L'énoncé découle du Lemme 2.22. □

Les Lemmes 2.21 et 2.23 prouvent le théorème suivant.

Théorème 2.24. *L'Algorithme NSUC est non-trivial.*

Intervalle-solo-rapidité non-triviale

Dans cette section nous prouvons que notre algorithme est intervalle-solo-rapide non-trivialement. Informellement, cela veut dire que durant l'exécution d'une opération un processus effectue des primitives CAS seulement si cette opération est concurrence d'intervalle. De plus, une opération restant incomplète à cause d'un processus fautif, ne cause pas l'utilisation infinie de primitives autre que le *Read* et $Write$. Ce sont les même éléments que pour la propriété de non-trivialité qui sont utilisés pour cette propriété, les démonstrations vont se faire de la même façon.

Observation 2.25. *Un processus p_i effectue des primitives autre que Read et Write pendant l'exécution d'une opération op seulement si il lit $LEVEL_A(i) \geq seq^i_{op} + 1$ à un moment donné au cours de l'exécution de op.*

Cela provient du fait qu'un processus effectue des primitives autre que *Read* et $Write$ seulement aux lignes 28 et 29. Et, pour exécuter ces lignes, il a du exécuter les lignes 13 ou 21.

Le lemme suivant se démontre à l'aide de l'observation 2.25 et du Lemme 2.19, de manière similaire au Lemme 2.21 :

Lemme 2.26. *Pour toute exécution donnée α, soit p un processus qui effectue des primitives autre que Read et Write pendant l'exécution d'une instance d'opération op. Alors op est concurrente à une autre opération op' en α.*

Le lemme suivant se démontre à l'aide de l'observation 2.25 et du Lemme 2.22, de manière similaire au Lemme 2.23 :

Lemme 2.27. *Fixons une exécution α. Soit p_i un processus fautif en α pendant l'exécution d'une opération op. Pour tout processus p_j, définissons op_1, op_2 et op_3 comme trois opérations consécutives exécutées par p_j de manière concurrente à op. p_j effectue des primitives autre que Read et Write lors de l'exécution de op_3 seulement s'il existe une opération $op' \neq op$ telle que l'invocation de op' suit la réponse de op_1 et que op' soit concurrente avec op_3.*

Informellement, ce lemme établit qu'une opération *op* qui ne finit pas peut entraîner l'application de primitives autre que *Read* et $Write$ lors de l'exécution d'au plus deux opérations concurrentes avec *op* pour un processus.

Le Lemme 2.26 et le Lemme 2.27 prouvent le théorème suivant.

Théorème 2.28. *L'Algorithme NSUC est intervalle-solo-rapide non-trivialement.*

Linéarisabilité

Soit une exécution α, pour toute configuration $C \in \alpha$, par le Lemme 2.14 et le Lemme 2.9, $\forall 0 < v \leq seq_O(C)$ il y a un unique état *state* tel qu'un

processus a écrit $< v, -, state >$ dans OC ou $< v, state >$ dans OS avant la configuration C. Nous disons qu'une opération qui change l'état de l'objet par l'état $state$ tel qu'un processus a écrit $< v, -, state >$ dans OC ou $< v, state >$ dans OS, est associé à la séquence v et aussi que l'état $state$ est associé à v.

Soit π une permutation des opérations de haut niveau en α. Nous construisons π en distinguant trois types d'opérations : Les opérations de lecture seule, les opérations qui ne sont pas des lectures seules et qui renvoient une réponse $res \neq \perp$ et enfin les opérations qui ne sont pas des lectures seules et qui renvoient $\perp$.

Premièrement, les opérations qui ne sont pas des lectures seules et retournent $res \neq \perp$ changent l'état de l'objet, elles sont donc associées à une valeur de séquence. Nous ordonnons ces opérations dans l'ordre croissant de la valeur de séquence qui leur est associée.

Deuxièmement, nous considérons chaque opération de lecture seule dans l'ordre pour laquelle la réponse a lieu dans α. Une opération de lecture seule op est placée immédiatement avant l'opération associée à la valeur de séquence $seq_{op} + 1$.

Finalement, une opération op qui n'est pas une lecture seule et qui retourne $\perp$ est placée immédiatement avant l'opération associée à la valeur de séquence $seq_{op} + 1$.

Observation 2.29. *Soit op et op' des opérations de α. Par construction de π, si $seq_{op} < seq_{op'}$ alors op précède op' dans π.*

Observation 2.30. *Soit une exécution fixée α, pour toutes configurations $C \in \alpha$ et pour tous $v \leq seq_O(C)$, par construction de π, la dernière opération op de π tel que $seq_{op} = v$ est l'opération associée à la valeur de séquence $seq_{op} + 1$.*

Lemme 2.31. *Soit s l'état de l'objet de type T^{da} avant l'opération op dans π. Alors la réponse res retournée par op dans α est telle que $(s, op, s', res) \in \Delta$ où Δ est la spécification séquentielle de T^{da}.*

Démonstration. Soit Δ la spécification séquentielle de T^{da}. Si op est une opération qui n'est pas de lecture seule et retourne $\perp$, alors par définition du type T^{da}, pour tout état s de l'objet $(s, op, s, \perp) \in \Delta$.

Si op est une opération qui n'est pas de lecture seule et retourne $res \neq \perp$, alors elle est associée à une valeur de séquence v et $seq_{op} = v - 1$. Donc res est telle que $(s, op, s', res) \in \Delta$ avec s l'état associé à la valeur de séquence $v - 1$ ou à l'état initial si $v = 1$. Par l'Observation 2.29, l'état précédent est l'état associé à $v - 1$ ou l'état initial si $v = 1$. L'énoncé est vrai.

Si op est une opération de lecture seule, alors res est tel que $(s, op, s, res) \in \Delta$ avec s l'état associé à la valeur de séquence seq_{op} ou à l'état initial si $seq_{op} = 0$. Par les Observations 2.29 et 2.30, l'état précédent est l'état associé à seq_{op} ou l'état initial si $seq_{op} = 0$. Donc l'énoncé est vrai. □

Lemme 2.32. *Soit op et op′ deux opérations de haut niveau dans α. Si l'invocation de op′ suit la réponse de op dans α, alors op précède op′ dans π.*

Démonstration. Soit l'invocation d'une opération op' faite après la réponse d'une opération op dans α.

Premièrement, supposons que op n'est pas de lecture seule. Alors d'après le Lemme 2.18, les Observations 2.3 et 2.4, $seq_{op} < seq_{op'}$. Donc par l'Observation 2.29, op précède op' dans π.

Maintenant, supposons que op est une opération de lecture seule. Alors par les Observations 2.3 et 2.4, $seq_{op} \leq seq_{op'}$. Si $seq_{op} < seq_{op'}$, d'après l'Observation 2.29, op précède op' dans π. Considérons le cas où $seq_{op} = seq_{op'}$. Deux cas peuvent être distingués. Premièrement, considérons que op' est également une opération de lecture seule. La réponse de op a lieu en α avant la réponse de op', donc par construction de π pour les opérations de lecture seule, op précède op'. Finalement, considérons que op' ne soit pas une opération de lecture seule. Par construction de π, elle est placée après toutes les opérations de lecture seule avec la même valeur de séquence seq_{op}. Nous avons aussi que op précède op' dans π. □

Le Lemme 2.31 et le Lemme 2.32 prouvent le théorème suivant.

Théorème 2.33. *L'algorithme NSUC est linéarisable.*

2.3 Résumé

Dans ce chapitre, nous avons présenté nos résultats sur le mécanisme de construction universelle. Nous avons prouvé une borne inférieure linéaire au nombre de processus sur la complexité en espace de l'implémentation d'objets déterministiquement abandonnables intervalle-solo-rapide non-trivialement qui assure la propriété de non-trivialité (vis-à-vis de la propriété de l'abandon). Pour cela, nous avons adapté la notion d'objet perturbable de Jayanti *et al.* [2000] aux objets déterministiquement abandonnables. Nous avons également prouvé qu'un objet k-CAS déterministiquement abandonnable est perturbable selon cette définition.

Nous avons présenté ensuite une construction universelle intervalle-solo-rapide non-trivialement pour des objets abandonnables de manière déterministique avec une propriété de non-trivialité, appelée NSUC. Toute implémentation issue de notre construction universelle a une complexité en espace asymptotique optimale si l'objet implémenté est perturbable et si son implémentation séquentielle nécessite un nombre constant de registres. L'algorithme NSUC garantit que les opérations qui ne modifient pas l'objet implémenté ne retourne jamais de réponse "abandon". De plus, en cas de concurrence d'intervalle, au moins une opération d'écriture réussit à modifier l'état de l'objet. Nous assurons également que si un processus tombe en panne lors d'une exécution, alors

celui-ci ne cause l'abandon que d'au plus deux opérations par processus. Notre construction universelle utilise $O(n)$ objets de base et elle stocke au plus $2n+1$ versions de l'objet.

Chapitre 3

Abstractions de base pour résoudre le consensus et le k-accord

Pour résoudre un problème complexe, il est classique de le découper en sous-problèmes. C'est pourquoi de nombreux objets de haut niveau sont définis dans le but d'être utilisés pour implémenter un autre objet plus complexe et/ou d'en comprendre la complexité.

Ainsi dans les travaux proposés pour dépasser l'impossibilité de l'implémentation du consensus sans attente avec seulement des registres, plusieurs abstractions ont été proposées. De même, au cours de nos travaux sur le consensus et le k-accord (voir le chapitre suivant), nous avons été amenés à introduire un nouvel objet : le k-value-splitter.

Nous donnons dans ce chapitre une définition du k-value-splitter et plusieurs implémentations. Puis, nous le comparons aux principaux objets préexistants proposés comme blocs de base pour résoudre l'accord ou des problèmes de coordination proches. Cela nous permet ainsi de mieux comprendre les différences entre ces objets (propriétés et complexité) et, dans certains cas, d'améliorer la complexité en espace des implémentations des objets préexistants (pour le conflict-detector et le k-converge).

La définition du k-value-splitter et ses implémentations ont été publiés dans l'article Capdevielle *et al.* [2015]. L'amélioration de l'implémentation du conflict-detector a été publiée dans Capdevielle *et al.* [2016].

3.1 k-value-splitter

La définition du k-value-splitter s'est construite à partir du splitter (Lamport [1987], Moir et Anderson [1995], Buhrman *et al.* [1995]), mais avec la volonté de ne pas avoir à utiliser l'identifiant des processus pour pouvoir l'im-

plémenter dans un contexte anonyme. Nous détaillons ce point dans la sous-section 3.2.1.

Un k-value-splitter possède une seule opération $split(v)$, avec $v \in V$, pour V un certain ensemble d'entrées et relaxe la propriété (2) des splitters en permettant à plusieurs processus d'obtenir la sortie *true* mais pour pas plus de k valeurs d'entrée différentes.

Plus précisément :

Définition 3.1. Un k-value-splitter fournit une seule opération $split(v)$, prenant en entrée une valeur v d'un ensemble V, et retourne un booléen et pour toutes les exécutions assure les propriétés suivantes :

1. **Exécution solo** Si une opération *split* termine avant que toute autre opération *split* soit invoquée, le booléen *true* est renvoyé pour cette opération.
2. **k-VS-Accord** Soit S l'ensemble des entrées pour lesquelles une opération *split* a retourné *true*, alors $\mid S \mid \leq k$

Nous présentons deux implémentations d'un k-value-splitter anonyme. La première des deux est invariante-aux-entrées et a une complexité en espace de $O(\sqrt{n/k})$, quelque soit le nombre m d'entrées possibles. La seconde a une complexité en espace de $O(\log(m/k)/\log\log(m/k))$, quel que soit le nombre de processus n.

Dans le prochain chapitre, nous utilisons cette abstraction pour implémenter le k-accord et nous démontrons l'optimalité de la complexité en espace des implémentations de cette abstraction.

3.1.1 k-value-splitter invariant-aux-entrées

L'Algorithme 7 décrit notre implémentation anonyme et invariante-aux-entrées d'un k-value-splitter. L'algorithme utilise seulement un tableau R de b registres, où $b^2 - 3b + 4 > (2n-2)/k$, il est donc trivialement intervalle-solo-rapide.

Dans l'algorithme, un processus p réalisant une opération $split(v)$ essaye d'écrire sa valeur d'entrée dans chacun des registres $R[0], \ldots, R[b-1]$ dans l'ordre. A chaque fois, avant d'écrire dans $R[i]$, p lit $i+1$ registres pour vérifier que $R[0], \ldots, R[i-1]$ stockent sa valeur v et que $R[i]$ stocke la valeur initiale $\perp$. Si ce n'est pas le cas, l'opération retourne $false$. Sinon, après la dernière écriture dans $R[b-1]$, l'opération retourne *true*.

Étant donné que de $i = 0$ à $b-1$, dans la i^{me} itération, un processus lit i registres, alors l'algorithme a une complexité en pas de calcul de $O(n/k)$. Ce qui a été démontré comme asymptotiquement optimal par Aspnes et Ellen [2014] dans le cas $k = 1$.

Variables partagées :
Tableau de registres $R[0 \dots b-1]$ avec $b^2 - 3b + 4 > 2(n-1)/k$. Initialement $\perp$

Procédure : $split(v)$

```
1  Lastwritten := −1 ;
2  while (Lastwritten < b − 1) do
3      i := 0 ;
4      while (i ≤ Lastwritten) do
5          if Read(R[i]) ≠ v then return false
6          i + + ;
7      end
8      if Read(R[Lastwritten + 1]) ≠ ⊥ then return false;
9      Lastwritten + + ;
10     Write(R[Lastwritten], v) ;
11 end
12 return true;
```

Algorithme 7: k-value-splitter anonyme et invariant-aux-entrées

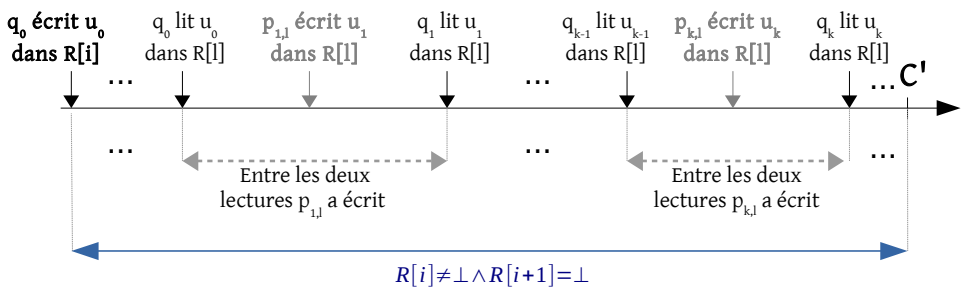

FIGURE 3.1 – Illustration d'une partie de l'exécution γ du Lemme 3.1

Preuves

Le lemme suivant est utilisé pour démontrer que l'Algorithme 7 satisfait la propriété de k-VS-Accord.

Lemme 3.1. *Pour toute exécution γ, dans laquelle $k+1$ processus q_j écrivent dans $R[i]$ et $R[i+1]$ (pour un $0 < i < b-1$) $k+1$ valeurs différentes u_j (avec $j \in [0,k]$), il y a un ensemble P_i d'au moins $i \cdot k$ processus (différents de q_i) tel que : (1) À la configuration succédant immédiatement la dernière écriture exécutée par un processus de P_i, $R[i+1] = \perp$; (2) γ passe par une configuration C dans laquelle $R[i] \neq \perp$ et chaque processus de P_i exécute exactement une écriture après C.*

Démonstration. Fixons i tel que $0 < i < b-1$ et soit $Q = \{q_0, \dots, q_k\}$, $k+1$ processus distincts qui écrivent respectivement les valeurs distinctes $\{u_0, \dots, u_k\}$, dans les registres $R[i]$ et $R[i+1]$ lors de l'exécution γ.

D'après le pseudo-code de l'Algorithme 7, avant d'écrire dans $R[i+1]$, un processus s'assure que les registres $R[0], R[1], \ldots, R[i]$ contiennent sa propre valeur, et que $R[i+1]$ contient la valeur initiale $\perp$.

Soit C' la configuration de γ, immédiatement après que le dernier processus de Q effectue sa lecture de $R[i+1]$ avant d'y écrire. D'après l'algorithme, toutes les écritures dans $R[i+1]$ par n'importe quel processus q_j suivent C' dans γ.

Considérons l'ordre dans lequel les processus de Q ont effectués la primitive de lecture dans $R[l]$, avec $0 \leq l < i$ après leur écriture dans $R[i]$. Supposons, sans perte de généralité, que cet ordre est $q_0, \ldots, q_k$. Comme pour chaque $h = 1, \ldots, k$, le processus q_{h-1} lit la valeur u_{h-1} dans le registre $R[l]$ et que q_h y lit la valeur u_h, entre ces deux lectures il doit y avoir une écriture par un processus $p_{h,l}$.

Nous montrons que c'est la dernière écriture de $p_{h,l}$ dans γ. En effet, avant d'effectuer une nouvelle écriture (sur $R[l+1]$), $p_{h,l}$ lit tous les registres, et en particulier il lit $R[i]$, où $i > l$. Comme l'écriture par $p_{h,l}$ suit la lecture de $R[l]$ par le processus q_{h-1}, et que d'après l'algorithme, q_{h-1} a écrit auparavant dans $R[i]$ et donc dans $R[l+1]$, dans la configuration immédiatement avant l'écriture dans $R[l]$ par $p_{h,l}$, $R[l+1] \neq \perp$. La vérification à la ligne 8 implique que $p_{h,l}$ ne peut pas écrire dans un autre registre après son écriture de $R[l]$.

Nous avons donc que $p_{h,l}$ est différent pour tout $p_{h',l'}$ si $h' \neq h$ ou si $l \neq l'$. Donc, il y a $i \cdot k$ tels processus : un pour chaque registre $R[l]$, $0 \leq l < i$ et pour chaque h, $1 \leq h \leq k$. De plus, tous les processus $p_{h,l}$ sont distincts des q_j : cela contredirait sinon le fait que q_j écrit dans $R[i]$, $i > l$.

Enfin, comme la dernière écriture de $p_{h,l}$ précède la configuration C', à la configuration immédiatement après cette écriture, $R[i+1]$ contient toujours sa valeur initiale. Une illustration est donnée dans la Figure 3.1. Ainsi, l'ensemble P_i des ik processus $p_{h,l}$, $l = 0, \ldots, i-1$, et $h = 1, \ldots k$, satisfont les deux conditions du lemme. □

Lemme 3.2 (k-VS-accord)**.** *Soit S l'ensemble des entrées pour lesquelles une opération split a retourné true, alors $\mid S \mid \leq k$*

Démonstration. Soit S l'ensemble des entrées pour lesquelles une opération *split* a retourné *true*. Par contradiction, supposons que $|S| > k$ dans une exécution E. Donc, $k+1$ invocations de *split* effectuées par les processus $q_0, \ldots, q_k$ ont retourné *true* dans E. Étant donné qu'un processus q_j doit écrire sa valeur d'entrée dans les b registres avant de retourner *true*, alors pour chaque $0 \leq i \leq b-1$ et $0 \leq j \leq k$, q_j a écrit dans le registre $R[i]$ la valeur u_j. Pour chaque $i = 1, \ldots b-2$, soit P_i les $i \cdot k$ processus, qui satisfont les conditions spécifiées par le Lemme 3.1.

Soit deux ensembles P_i, P_l, avec $0 < i < l < b-1$. Par la définition de P_i, dans la configuration suivant la dernière écriture des processus de P_i, $R[i+1]$ contient la valeur initiale, et donc d'après l'algorithme, pour tous $l > i$, $R[l]$ également. Par la définition de P_l, chaque processus de P_l a exécuté une

opération d'écriture après une configuration dans laquelle $R[l] \neq \perp$. Ainsi P_i et P_l sont disjoints.

Rappelons que chaque q_j, $j \in \{0, \ldots, k\}$, écrit dans $R[b-1]$ et donc n'appartient pas à $\cup_{i=1}^{b-2} P_i$. Finalement, le nombre total de processus nécessaires à l'exécution E est d'au moins $k+1+\sum_{i=1}^{b-2} ik = k+1+k\frac{b^2-3b+2}{2}$, ce qui contredit l'hypothèse que $b^2 - 3b + 4 > \frac{2n-2}{k}$. □

Théorème 3.3. *L'Algorithme 7 est une implémentation anonyme, intervalle solo-rapide et invariante-aux-entrées d'un k-value-splitter avec une complexité en espace de* $O(\sqrt{n/k})$.

Démonstration. Étant donné que seulement des registres sont utilisés et que l'opération *split* est sans attente, l'algorithme est intervalle-solo-rapide.

D'après le Lemme 3.2, l'algorithme satisfait la propriété de *k-VS-accord*. Nous prouvons dans la suite que la propriété de *solo exécution* est également satisfaite. Par contradiction, supposons γ une exécution dans laquelle une opération *split*, par un processus p, complète avant que toute opération soit invoquée, et que le booléen $false$ est renvoyé pour cette opération. Par l'inspection du pseudo-code, il est aisé de voir que la valeur contenu dans la variable *Lastwritten* est égale à l'index de la dernière écriture de p ou -1 si aucune écriture n'a encore eu lieu. Ainsi, pour retourner $false$, p a soit lu une valeur différente de sa propre valeur (test de la ligne 5), soit une valeur différente de $\perp$ dans un registre qu'il n'a pas encore écrit (test de la ligne 8). Dans les deux cas, cela contredit le fait que dans γ l'exécution de p s'est finie avant toute autre opération. Ainsi, l'algorithme satisfait la propriété de *solo exécution*. □

3.1.2 k-value-splitter non-invariant-aux-entrées

Variables partagées :
Registres $R[1..b]$, initialement $\perp$

Procédure : *split*(v), avec $v \in V_i$

```
1 for i := 1..b do
2     t := Read(R[π_{V_i}(i)]);
3     if t = ⊥ then Write(R[π_{V_i}(i)], v);
4     if t ≠ v then return false;
5 end
6 return true;
```

Algorithme 8: k-value-splitter non-invariant-aux-entrées

Nous décrivons maintenant l'algorithme anonyme du k-value-splitter dont la complexité en espace est de $O(\log(m/k)/\log\log(m/k))$. L'algorithme se base sur le travail de Aspnes et Ellen [2014]. On généralise l'algorithme du

conflict-detector introduit dans ce papier pour le transformer en un k-value-splitter (voir Algorithme 8). Pour cela, au lieu d'associer à chaque valeur d'entrée v une permutation π_v des registres, on associe k valeurs à une permutation, le reste étant inchangé.

Plus précisément, l'algorithme utilise un tableau R de b registres, où $b! = \lceil m/k \rceil$. L'ensemble des valeurs d'entrée est partitionné en $l = \lceil m/k \rceil$ sous-ensemble V_1, ..., V_l ; la taille de chaque sous-ensemble étant au plus de k. Une unique permutation π_{V_i} des registres de R est associé à chaque sous-ensemble V_i, $i \in \{1 \dots l\}$; de telle sorte qu'un processus pendant l'exécution d'une opération $split(v)$, avec $v \in V_i$ accède aux registres dans l'ordre de la permutation π_{V_i}. Par conséquent, l'algorithme n'est pas invariant-aux-entrées. Lors du j^{me} accès, un processus exécutant $split(v)$ $v \in V_i$, commence par lire le registre $R[\pi_{V_i}(j)]$; si la valeur initiale est lue, le processus écrit v dans le registre, par contre, si une valeur différente de la sienne est lue, il retourne $false$ (un conflit est détecté). Finalement, si un processus écrit sa valeur dans tous les registres, il retourne $true$. L'algorithme est également trivialement anonyme et intervalle-solo-rapide.

Théorème 3.4. *L'Algorithme 8 implémente un k-value-splitter m-valué anonyme et intervalle-solo-rapide avec une complexité en espace de $O(\log(m/k)/\log\log(m/k))$.*

Démonstration. Si une opération $split(v)$ s'exécute seule, alors aucune autre valeur que v ne peut être lue (à la ligne 2). Donc, la propriété d'*exécution solo* est assurée.

Supposons par contradiction que $k+1$ opérations distinctes avec des valeurs en entrée distinctes retournent $true$. Parmi les $k+1$ valeurs, il y en a deux distinctes, v et v' qui appartiennent à deux sous-ensembles distincts, V et V'. Soient j, ℓ deux index dans $\{1, \dots, b\}$ tel que j apparaît avant ℓ dans π_V mais ℓ apparaît avant j dans $\pi_{V'}$. D'après le pseudo-code de l'algorithme, avant de retourner $true$, p_v et $p_{v'}$ ont lu respectivement, v et v' dans les registres $R[j]$ et $R[\ell]$.

Sans perte de généralité, supposons que v soit écrit dans $R[j]$ avant que v' le soit dans $R[\ell]$. Avant qu'aucun des processus n'exécutant $split(v')$ lisent $R[j]$ à la ligne 2 (et donc écrivent v' dans $R[j]$ à la ligne 3), v' a été écrit dans $R[\ell]$, et par hypothèse, v a été écrit dans $R[j]$. Par conséquent, ces processus ne peuvent pas lire $\perp$ dans $R[j]$ et n'écriront pas dans $R[j]$, ce qui est une contradiction. Ainsi, l'algorithme satisfait la propriété de *k-VS-accord*.

Étant donné que chaque opération effectue b écritures et b lectures au plus, avec $b! = \lceil m/k \rceil$, la complexité en pas de calcul et en espace est de $O(\log(m/k)/\log\log(m/k))$.

□

3.2 Étude comparée

Le k-value-splitter est proche du splitter, de plus, nous avons vu que l'implémentation du k-value-splitter peut se faire à partir d'un algorithme de conflict-detector, voir même en utilisant le conflict-detector dans le cas où $k = 1$. D'autres objets ont été également définis pour résoudre le consensus et le k-accord.

À l'aide de cette nouvelle abstraction, nous allons dans cette section comparer de manière plus précise ces objets entre eux, en commençant par une comparaison avec le 1-value-splitter, puis avec sa généralisation, le k-value-splitter (avec $k > 1$). Dans certains cas, nous proposons une amélioration de l'implémentation des objets étudiés. Un résumé des objets comparés et des relations est présenté dans la Figure 3.2.

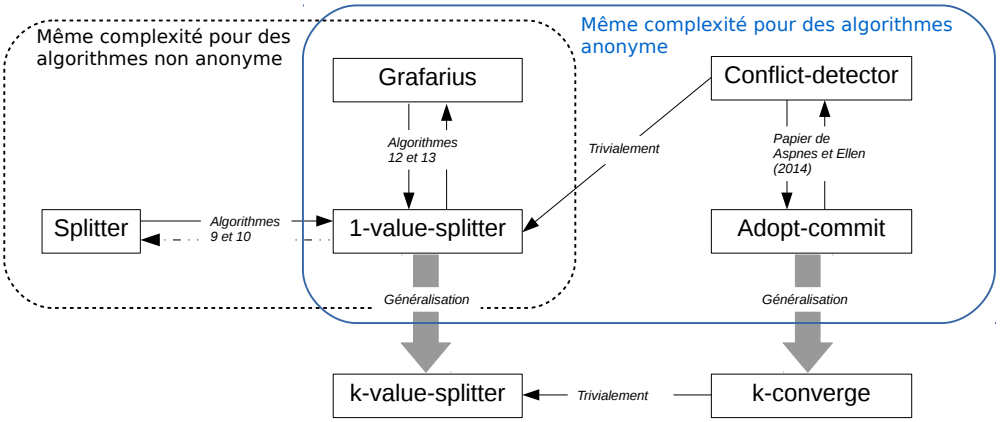

FIGURE 3.2 – Illustration des relations entre le k-value-splitter et le 1-value-splitter et les autres objets utilisés pour résoudre le consensus et le k-accord

3.2.1 1-value-splitter versus Splitter

Un splitter fournit aux processus une seule opération *split*(). L'opération retourne un booléen tel que : (1) si un processus exécute seul et finit avant que n'importe quel autre processus commencent, alors il obtient *true* et (2) *true* est retourné à au plus un processus.

Le splitter assure les propriétés du 1-value-splitter, ainsi l'implémentation d'un 1-value-splitter à partir d'un splitter est trivial (voir l'Algorithme 9). Dans l'autre sens, le 1-value-splitter ne peut assurer les propriétés du splitter que si les arguments en entrée des processus sont tous différents, ce qui est trivial à réaliser dans un contexte non-anonyme (voir Algorithme 10). Les algorithmes 9 et 10 démontrent que les algorithmes non-anonymes qui implémentent le splitter ou le 1-value-splitter ont la même complexité spatiale et en

temps. Et comme l'objet splitter peut être implémenté par des registres avec une complexité constante, en pas de calcul et en espace, le 1-value-splitter l'est aussi.

Variables partagées :
Un splitter S

Procédure : *split*(v)
1 **return** S.split();

Algorithme 9: 1-value-splitter

Variables partagées :
Un 1-value-splitter VS

Procédure : *split*()
1 **return** VS.split(p);

Algorithme 10: splitter, opération faite par un processus p

Cependant, dans un contexte anonyme, l'équivalence n'est plus valable. Contrairement au 1-value-splitter implémentable dans un contexte anonyme avec seulement des primitives *Read* et *Write*, comme démontré avec les Algorithmes 7 et 8, l'objet splitter est impossible à implémenter avec seulement des primitives *Read* et *Write* (on peut le déduire de l'équivalence donné par Guerraoui et Ruppert [2005]).

En effet, Guerraoui et Ruppert [2005] prouvent qu'il faut que le type T d'un objet soit *idempotent* pour qu'il puisse être implémenté sans obstruction, de manière anonyme et uniquement avec des registres. Pour définir l'*idempotence* d'un type T, nous donnons au préalable plusieurs définitions.

Une *histoire séquentielle* est une séquence d'événement, chaque événement étant une paire consistant en l'invocation d'une opération et sa réponse. Une histoire séquentielle est *légale* (pour un état initial donné) si elle est consistante avec la spécification du type de l'objet. Deux histoires séquentielles H et H' sont *équivalentes* si, pour toute histoire séquentielle G, la concaténation de H et G, notée HG, est légale si et seulement si $H'G$ est légale. Un événement e est indempotent si, pour toutes histoires séquentielles H, si He est légale, alors Hee est légale et équivalente à He. Enfin, un objet est dit *idempotent* si toutes ses opérations sont idempotentes.

Lemme 3.5. *Un objet splitter n'est pas idempotent.*

Démonstration. Soit H une histoire séquentielle tel qu'aucune opération n'a été effectuée sur le splitter et e l'événement qui a comme invocation *split*() et comme réponse *true*. He est légale car la spécification du splitter permet le retour *true* à l'invocation de *split*() sur l'état initial de l'objet. Cependant,

Hee n'est pas légale. En effet, la spécification du splitter limite le retour $true$ à une seule invocation. □

Le type splitter n'est pas idempotent et donc il y a bien impossibilité d'implémenter un splitter avec seulement des registres dans un contexte anonyme. De manière intuitive, ce résultat n'est pas étonnant, en effet sans l'utilisation d'identifiants ou de primitives plus fortes que $Read$ et $Write$, comment empêcher des clones, des processus faisant exactement la même chose, de se détecter et donc de respecter la propriété de retourner $true$ à un seul processus ?

3.2.2 1-value-splitter versus conflict-detector et adopt-commit

L'objet adopt-commit, qui provient du protocole $adopt - commit$ proposé par Gafni [1998], supporte une seule opération $adoptCommit(u)$ avec u une valeur d'entrée venant d'un ensemble U. L'opération retourne une sortie de la forme $(flag, v)$ où $flag \in \{adopt, commit\}$ et le second élément v appartient à U. De plus, un objet adopt-commit respecte les propriétés suivantes :

Validité La valeur de sortie d'une opération est la valeur d'entrée d'une opération (potentiellement d'une autre opération).

Cohérence Si la sortie d'une opération est $(commit, v)$, alors toutes les sorties sont soit $(commit, v)$, soit $(adopt, v)$.

Convergence Si toutes les valeurs d'entrée sont v, alors toutes les sorties doivent être $(commit, v)$.

Pour prouver des bornes sur l'objet adopt-commit, Aspnes et Ellen [2014] ont introduit l'objet conflict-detector. Il fournit une opération $check(v)$ avec v une valeur d'entrée et retourne un booléen : $true$ pour indiquer un conflit ou $false$ pour indiquer l'absence de conflit. Il assure également les deux propriétés suivantes :

Pour toute exécution qui contient une opération $check(v)$ et une opération $check(v')$ avec $v \neq v'$, alors au moins l'une de ces opérations retourne $true$.

Si toutes les valeurs d'entrée sont les mêmes, alors toutes les sorties sont $false$.

Aspnes et Ellen [2014] ont démontré que l'on peut implémenter un objet adopt-commit à l'aide d'un nombre constant de registres et d'un objet conflict-detector et que l'implémentation d'un objet conflict-detector se fait simplement à l'aide d'un objet adopt-commit. Ces implémentations n'utilisant pas les identifiants des processus, cette équivalence est donc valable dans un contexte anonyme ou non.

Les propriétés du 1-value-splitter sont un sous ensemble des propriétés qu'assure le conflict-detector et de même pour l'adopt-commit. Le 1-value-splitter n'assure pas que si toutes les valeurs d'entrée sont identiques, la réponse $true$ sera renvoyée à tous, contrairement à l'adopt-commit qui assure que la réponse est $(commit, v)$ pour tous et le conflict-detector qui assure le retour $false$ pour tous. La question est de savoir si cette propriété non assuré par le 1-value-splitter a une répercussion sur les complexités de l'implémentation de ces objets. Nous allons voir que c'est le cas dans un contexte non-anonyme mais non dans le contexte anonyme.

Dans un contexte non-anonyme, nous avons établi que le 1-value splitter est implémentable à l'aide de registres avec une complexité en pas de calcul constant. Comme Aspnes et Ellen [2014] prouvent une borne inférieur de $\Omega(min(\frac{logm}{loglogm}), \frac{\sqrt{logn}}{loglogn})$ pour l'objet adopt-commit, sur le nombre de pas que doit faire un processus, avec m le nombre de valeurs d'entrée différentes et n le nombre de processus, on en conclut que l'objet 1-value-splitter n'est pas équivalent en terme de complexité en pas de calcul aux objets conflict-detector et adopt-commit, dans le contexte non-anonyme. La propriété supplémentaire a un coût dans un contexte non-anonyme.

Dans un contexte anonyme, nous avons établi que la complexité du 1-value splitter n'est plus constante, que se soit en pas de calcul ou en espace ; et finalement le coût de la propriété supplémentaire du conflict-detector vis-à-vis du 1-value-splitter est nul, dans ce contexte. En effet, nous présentons un algorithme (l'Algorithme 11) implémentant le conflict-detector avec des complexités asymptotiquement identiques aux complexités des algorithmes du 1-value-splitter qui étaient eux même asymptotiquement optimaux. Nous améliorons ainsi l'algorithme avec une complexité linaire en espace proposé par Aspnes et Ellen [2014], en refermant la question resté alors ouverte de la borne optimale en espace.

Algorithme de conflict-detector asymtotiquement optimal

Description L'Algorithme 11 est similaire à celui du value-splitter invariant-aux-entrées présenté dans la Section 3.1. Mais il garantit également la propriété de convergence de l'adopt-commit. Pour assurer cette propriété, un processus qui détecte un autre processus étant plus avancé dans le calcul avec la même valeur d'entrée ne retourne pas $true$ mais avance directement à l'écriture du premier registre contenant encore la valeur initiale. Avant d'écrire, il lit l'ensemble des registres déjà écrits pour vérifier qu'ils contiennent toujours sa valeur d'entrée.

Preuves

Lemme 3.6. *Si lors d'une exécution E, deux valeurs différentes v et w sont*

Variables partagées :
Tableau de registres $R[0 \ldots b-1]$ avec $b^2 - 4 > 4n$ et $b \geq 4$. Initialement$\perp$

Procédure : *check-value*(v)

```
1  i ← 0;
2  next_writing ← 0;
3  while i < b do
4      i ← 0;
5      res ← Read(R[i]);
6      while i < b ∧ res ≠ ⊥ do
7          if res ≠ v then return true ;
8          i++;
9          if i ≠ b then res ← Read(R[i]);
10     end
11     if i < b ∧ i = next_writing then
12         Write(R[i], v);
13         next_writing ← i + 1;
14     else
15         next_writing ← i;
16     end
17 end
18 return false;
```

Algorithme 11: Conflict-detector invariant-aux-entrées

écrites dans $R[i]$ et dans $R[i+1]$ pour un $0 \leq i < b-1$, alors il y a un ensemble P_i de $i+1$ processus qui satisfont les propositions suivantes :

- *(1) Tous les registres du tableau R ayant des indices de 0 à i ont été écrits par un processus appartenant à P_i entre la première écriture dans $R[i]$ et la première écriture dans $R[i+1]$.*
- *(2) Après la première écriture dans $R[i+1]$, aucun processus de P_i n'écrit dans un registre avec un indice plus petit que $i+1$.*
- *(3) Chaque processus de P_i écrit une seule fois entre la première écriture dans $R[i]$ et la première écriture dans $R[i+1]$.*

Démonstration. Supposons que lors d'une exécution E, deux valeurs différentes sont écrites dans les registres $R[i]$ et $R[i+1]$, pour un $0 \leq i < b-1$. Soit p le premier processus à écrire dans $R[i+1]$ et soit v la valeur écrite alors. Soit q le premier processus à écrire dans $R[i+1]$ une valeur w telle que $w \neq v$.

Par le pseudo-code de l'Algorithme 11, après avoir mis sa variable locale *next_writing* à $i+1$ et avant d'écrire dans le registre $R[i+1]$, un processus lit les registres $R[0], R[1], \ldots, R[i+1]$, et la valeur lue dans $R[j]$ pour $0 \leq j \leq i$ est sa valeur d'entrée, tandis que c'est la valeur initiale pour $j = i+1$.

Considérons la séquence des opérations de lecture exécutées par p et q, respectivement, immédiatement après que leur variable locale *next_writing* soit mise à $i+1$ et avant leur écriture dans $R[i+1]$. Étant donné que pour chaque $j = 0, \ldots, i$, p lit v dans $R[j]$ et q lit w dans $R[j]$, il y a un processus p_j qui a écrit dans $R[j]$ entre les deux lectures. Soit $P_i \bigcup_{j=0}^{i} \{p_j\}$. Aucune variable locale *next_writing* n'est mise à $i+1$ avant la première écriture dans $R[i]$ (lignes 6, 8, 13 et 15). Donc, les processus de P_i ont écrit dans les registres $R[0] \ldots R[i]$ entre la première écriture dans $R[i]$ et la première écriture dans $R[i+1]$. La proposition (1) est satisfaite.

Maintenant nous prouvons qu'aucun processus de P_i écrira dans un registre avec un identifiant inférieur à $i+1$, après la première écriture dans $R[i+1]$, et que tous les processus de P_i écrivent une seule fois entre la première écriture dans $R[i]$ et la première écriture dans $R[i+1]$. Soit g un processus appartenant à P_i. Par définition g exécute au moins une écriture entre la première écriture dans $R[i]$ et la première écriture dans $R[i+1]$. Après la première de ces écritures, g lit les registres $R[0] \ldots R[h]$ avec $0 \leq h < b$ et la valeur lue dans $R[h]$ est soit $\perp$, soit une valeur différente de la valeur d'entrée de g. Dans le dernier cas, l'appel de *check*() par g retourne (ligne 7). Dans l'autre cas, $h > i$ et les primitives d'écriture suivantes, faites par g (s'il y en a) ne se feront pas sur les registres $R[0], \ldots R[i]$. De plus, g n'écrira pas dans les registres $R[i+1] \ldots R[b-1]$ entre la première écriture dans $R[i]$ et la première écriture dans $R[i+1]$. Les propositions (2) et (3) sont donc satisfaites. D'après les propositions (1) et (3), la taille de P_i est strictement plus grande que i. □

Observation 3.7. *Si lors d'une exécution E, deux valeurs distinctes v et w*

sont écrites dans $R[z+1]$*, alors les deux mêmes valeurs le sont dans* $R[z]$ *avec* $0 \leq z \leq b-2$.

Pour toute exécution E où deux valeurs distinctes sont écrites dans $R[i+1]$, on note $P_{i,j}$ l'ensemble des processus inclut dans P_i avec $0 \leq i, j \leq b-1$, qui écrivent dans les registres $R[j]$, entre la première écriture dans $R[i]$ et la première écriture dans $R[i+1]$.

Lemme 3.8. *Soit* E *une exécution dans laquelle deux valeurs distinctes sont écrites dans* $R[z+1]$*, avec* $z \in [0, b-2]$*. Pour tous* $i, j \in [0, z]$ *nous avons* $|P_{i,j}| \geq 1$ *si* $i \geq j$ *et* $|P_{i,j}| = 0$ *sinon.*

Démonstration. Si $i < j$, par définition de $P_{i,j}$ et étant donné que $R[j]$ ne peut pas être écrit avant la première écriture dans $R[i+1]$, nous avons que $|P_{i,j}| = 0$. Si $i \geq j$, $|P_{i,j}| \geq 1$ par la proposition (1) du Lemme 3.6. □

Lemme 3.9. *Soit* E *une exécution dans laquelle deux valeurs distinctes sont écrites dans* $R[z+1]$*, avec* $z \in [0, b-2]$*. Pour tous* $f \in [0, z]$*,* $j, j' \in [0, f]$ *et* $i, i' \in [f, z]$ *, si* $i \neq i'$ *ou* $j \neq j'$*, nous avons que* $P_{i,j} \cap P_{i',j'} = \emptyset$.

Démonstration. Sans perte de généralité, supposons que $i < i'$. Par la proposition (2) du Lemme 3.6, aucun processus de P_i, et donc aucun processus de $P_{i,j}$ n'écrit dans un registre avec un indice inférieur à $i+1$ après la première écriture dans $R[i+1]$. Étant donné que $j' < i+1$, aucun processus de $P_{i,j}$ n'écrit dans $R[j']$ après la première écriture dans $R[i+1]$. De plus, étant donné que $i' > i$, la première écriture dans $R[i']$ n'est pas faite avant la première écriture dans $R[i+1]$. Donc, aucun processus de $P_{i,j}$ n'appartient à $P_{i',j'}$.

Supposons maintenant que $i = i'$ et $j \neq j'$. Par la proposition (3) du Lemme 3.6, aucun processus de $P_{i,j}$ n'écrit dans $R[j']$ entre la première écriture dans $R[i]$ et celle dans $R[i+1]$. Donc, $P_{i,j} \cap P_{i',j'} = \emptyset$.

□

Notons S_i l'ensemble des processus qui écrivent dans $R[i]$ pendant une exécution E.

Lemme 3.10. *Soit* E *une exécution dans laquelle deux valeurs distinctes sont écrites dans* $R[z+1]$*, avec* $z \in [0, b-2]$*. Nous avons* $|\bigcup_{j=0}^{f} S_j| \geq (f+1)(z-f+1)$ *avec* f *un entier tel que* $0 \leq f \leq z \leq b-2$.

Démonstration. Par définition, $|\bigcup_{j=0}^{f} S_j| \geq |\bigcup_{j=0}^{f} \bigcup_{i=0}^{z} P_{i,j}|$. $|\bigcup_{j=0}^{f} \bigcup_{i=0}^{z} P_{i,j}| \geq |\bigcup_{j=0}^{f} \bigcup_{i=f}^{z} P_{i,j}|$.

D'après le Lemme 3.9, $|\bigcup_{j=0}^{f} \bigcup_{i=f}^{z} P_{i,j}| = \sum_{j=0}^{f} \sum_{i=f}^{z} |P_{i,j}|$.

D'après le Lemme 3.8, $\sum_{j=0}^{f} \sum_{i=f}^{z} |P_{i,j}| \geq (f+1)(z-f+1)$. □

Théorème 3.11. *Dans toute exécution qui contient check(v) et check(v') avec $v \neq v'$, au moins l'une des deux opérations retourne true.*

Démonstration. Supposons par contradiction qu'il existe une exécution E qui contient *check*(v) et *check*(v') avec $v \neq v'$ et que les deux opérations retournent $false$. D'après l'algorithme, deux processus ont écrit les valeur v et v' dans le registre $R[b-1]$.

Le nombre total de processus dans le système est plus grand que $|\bigcup_{j=0}^{b-2} S_j| \geq |\bigcup_{j=0}^{b/2} S_j|$ si $b \geq 4$. En appliquant le Lemme 3.10 avec $z = b-2$ et $f = \frac{b}{2}$, nous avons que $|\bigcup_{j=0}^{b/2} S_j| \geq (b/2+1)(b/2-1)$. Ce qui contredit le fait que $n < (b/2+1)(b/2-1)$ et $b \geq 4$. □

Théorème 3.12. *Dans une exécution dans laquelle toutes les opérations check ont la même valeur d'entrée, elles retournent toutes $false$.*

Démonstration. Cela provient du fait qu'un processus écrit dans les registres seulement sa valeur d'entrée et qu'il retourne *true* seulement s'il lit une valeur différente de sa propre valeur d'entrée. □

Théorème 3.13. *Pour toute exécution, toutes les opérations check faites par des processus corrects retournent.*

Démonstration. Si un processus lit une valeur différente de sa valeur d'entrée, il retourne à la ligne 7 et la propriété est prouvée. Considérons donc les processus qui lisent uniquement leur valeur d'entrée. À chaque itération de la boucle de la ligne 6, le processus incrémente de 1 la valeur de i, donc finalement le processus sort de cette boucle. Il reste à prouver qu'aucun processus n'effectue la boucle des lignes 3 à 16 indéfiniment. Dans la suite de la preuve nous nous référons toujours à cette boucle à moins de le spécifier.

Pour $k = 0, ..., b-1$ nous notons i_k et $next_writing_k$ les valeurs contenues respectivement, dans la variable locale i et dans la variable locale $next_writing$ à la fin de la k^{me} itération de la boucle par un processus p. Par la suite, nous prouvons que $\forall 0 \leq k < b-1$, nous avons $next_writing_k < next_writing_{k+1}$ et donc après $b+1$ itérations, la condition de la ligne 3 n'est plus vérifiée.

Par la ligne 6, $\forall 0 \leq j < i_k$ nous avons que $R[j] \neq \perp$ immédiatement avant que la ligne 11 soit exécutée dans la k^{me} itération de la boucle. Soit p écrit $R[i_k]$ et alors $next_writing_k$ est égal à $i_k + 1$ (lignes 12 et 13) ; soit p exécute la ligne 15 et $next_writing_k$ est égal à i_k. Dans les deux cas, $\forall 0 \leq j < next_writing_k$, nous avons que $R[j] \neq \perp$ à la fin de la k^{me} itération de la boucle.

Supposons que le processus p exécute la $k+1^{ieme}$ itération. Si ce n'est pas le cas, la preuve est faite. Alors, par la ligne 3, $next_writing_k < b$. Étant

donné que $\forall 0 \leq j < next_writing_k$, $R[j] \neq \perp$, nous avons que $i_{k+1} \geq next_writing_k$. Si $i_{k+1} = next_writing_k$, alors $i_{k+1} < b$. Donc p exécute la ligne 13 et $next_writing_{k+1} = i_{k+1} + 1 > next_writing_k$. Si $i_{k+1} > next_writing_k$, alors p exécute la ligne 15 et $next_writing_{k+1} = i_{k+1} > next_writing_k$. □

3.2.3 1-value-splitter versus grafarius

L'objet grafarius est un objet présenté par Sutra *et al.* [2014] pour implémenter une construction universelle et qui propose une opération $adoptCommit(u)$, avec u une valeur d'un ensemble V et retourne une paire $(flag, v)$, où $flag \in \{adopt, commit\}$ et le second élément v appartient à V. De plus, il satisfait les propriétés suivantes, pour toute exécution E et pour tout processus p qui invoque $adoptCommit(u)$ dans E :

Validité Si p retourne $(adopt, v)$, ou $(commit, v)$, alors un processus a invoqué une opération $adoptCommit(v)$ avant.

Cohérence Si p retourne $(commit, v)$, alors tous les processus retournent $(commit, v)$ ou $(adopt, v)$.

Solo convergence Si p retourne avant que tout autre processus n'invoque $adoptCommit$, alors p retourne $(commit, u)$.

Continuation Si un processus retourne avant que p n'invoque $adoptCommit$, p retourne $(adopt, v)$ ou $(commit, v)$, avec v une valeur proposée avant son invocation de $adoptCommit$.

De manière triviale, il est possible d'implémenter un 1-value-splitter à l'aide d'un grafarius, voir l'Algorithme 12. En effet, la propriété de solo convergence du grafarius va assurer la propriété d'exécution solo du 1-value-splitter ; tandis que les propriétés de cohérence et de validité du grafarius assurent la propriété de 1-VS-Accord du 1-value-splitter.

```
Variables partagées :
Un grafarius G

Procédure : value-splitter(v)
1 < flag, val > ← G.adoptCommit(v);
2 if flag = commit ∧ val = v then
3 |   return true;
4 else
5 |   return false;
6 end
```

Algorithme 12: 1-value-splitter

La construction inverse, présenté dans l'Algorithme 13, est moins évidente à cause de la propriété de continuation à respecter. Il découle des algorithmes 12

```
   Variables partagées :
   Un registre C, initialement false
   Un registre D, initialement ⊥
   Un 1-value-splitter VS
   Procédure : adoptCommit(v)
 1 if ¬VS.split(v) then
 2 |   C ← true;
 3 |   d ← D;
 4 |   if d ≠ ⊥ then return(adopt, d);
 5 |   D ← v;
 6 |   return (adopt, v);
 7 end
 8 if C then
 9 |   d ← D;
10 |   if d ≠ ⊥ then return(adopt, d);
11 end
12 D ← v;
13 if C then return(adopt, v);
14 return (commit, v);
```

Algorithme 13: Grafarius

et 13 que malgré la propriété de continuation du grafarius, les complexités asymptotiques du grafarius et du 1-value-splitter sont identiques (en anonyme et en non-anonyme). Nous donnons maintenant une description de l'Algorithme 13 et présentons les preuves de cette implémentation.

Algorithme du Grafarius à partir d'un 1-value-splitter

Description Pour effectuer l'opération adoptCommit du grafarius, la première action que fait le processus est d'exécuter l'opération *split* du 1-value-splitter avec la valeur d'entrée en argument.

Suivant le retour de l'opération sur le 1-value-splitter, deux cas se dégagent. Dans le premier cas le retour du 1-value-splitter est $false$ et le processus exécute les lignes 2 à 6. Il prévient qu'il y a potentiellement un conflit et lit dans D si une valeur a déjà été proposée pour l'adoption. Si une valeur est déjà présente, il la retourne ; sinon il écrit sa propre valeur dans D et retourne sa valeur. Lors de ces retours, c'est la valeur *adopt* qui est également retournée.

Dans le second cas, le retour du 1-value-splitter est $true$. Pour assurer la propriété de continuation du grafarius, le processus fait d'abord les lignes 8 à 10. Ainsi, si un processus a déjà signalé un potentiel conflit et qu'une valeur est présente dans la variable D, le processus retourne directement le couple *adopt* et la valeur dans D. Sinon, il écrit sa valeur dans D et la retourne, soit avec la valeur *commit* si aucun conflit potentiel n'a été détecté, soit avec la

valeur *adopt*.

Preuves Trivialement, la propriété de validité du grafarius est respectée :

Théorème 3.14 (Validité)**.** *Pour toute exécution E et pour tout processus p qui invoque $adoptCommit(u)$ dans E, si p retourne $(adopt, v)$, alors un processus a invoqué une opération $adoptCommit(v)$ avant.*

Pour démontrer la cohérence, nous commençons par la démonstration du lemme suivant :

Lemme 3.15. *Pour toute exécution E. Tous les processus qui retournent aux lignes 13 ou 14 dans E retournent la même valeur.*

Démonstration. Soit E une exécution, p un processus qui invoque $adoptCommit(u)$ et q un processus qui invoque $adoptCommit(u')$ dans E. Supposons que p et q retournent aux lignes 13 ou 14. Alors les invocations de $split(u)$ et $split(u')$ respectivement par p et q de la ligne 1 ont retourné *true*. Par la propriété de 1-VS-accord du 1-value-splitter, nous avons que $u = u'$. Donc, comme ils ont retourné leur propre valeur d'entrée, nous avons bien que p et q retournent la même valeur. □

Théorème 3.16 (Cohérence)**.** *Pour toute exécution E et pour tout processus p qui invoque $adoptCommit(u)$ dans E, si p retourne $(commit, v)$, alors tous les processus retournent $(commit, v)$ ou $(adopt, v)$.*

Démonstration. Soit E une exécution et p un processus qui invoque $adoptCommit(u)$ dans E. Supposons que p retourne $(commit, v)$.

Avant de retourner $(commit, v)$, p a d'abord écrit sa valeur dans D à la ligne 12, puis a lu $false$ dans C à la ligne 13. La variable C est initialement à $false$ et ne peut que être mise à *true*. Donc, tous les processus qui ont mis la variables C à *true* ne peuvent pas lire la valeur initiale dans D (puisque p a écrit dans D avant que C ne soit écrit). Ainsi, aucun processus ne peut exécuter les lignes 5 et 6.

Par conséquent, seuls les processus exécutant la ligne 12 peuvent écrire dans D. De plus, tous les processus qui exécutent cette ligne ont la même valeur d'entrée à cause de la propriété de 1-VS-accord du 1-value-splitter et du fait que l'invocation à cet objet leur ai retourné *true*. p fait partie de ces processus, donc D ne peut contenir que la valeur d'entrée de p, qu'il a retourné.

Soit un processus q qui invoque $adoptCommit(u')$. Nous avons vu qu'il ne peut pas retourner à la ligne 6. Si ce processus retourne aux lignes 4 ou 10, alors il retourne la valeur lue dans D qui est celle retournée par p. Si ce processus retourne à la ligne 13 ou à la ligne 14, alors, d'après le Lemme 3.15, la propriété est prouvée. □

Théorème 3.17 (Solo convergence)**.** *Pour toute exécution E et pour tout processus p qui invoque $adoptCommit(u)$ dans E, si p retourne avant que tout autre processus invoque $adoptCommit$, alors p retourne $(commit, u)$.*

Démonstration. Soit E une exécution et p un processus qui invoque $adoptCommit(u)$ dans E. Supposons que p retourne avant que tout autre processus n'invoque $adoptCommit$. Par conséquent, il invoque $split(u)$, et cette opération complète avant que toute autre opération sur le 1-value-splitter ne soit invoquée. Donc, par la propriété de solo exécution du 1-value-splitter, la réponse est $true$. Il n'exécute pas les lignes 2 à 6. Donc, lorsqu'il exécute la ligne 8, C contient sa valeur initiale $false$, de même lorsqu'il exécute la ligne 13, p retourne donc à la ligne 13, en retournant $(commit, u)$. □

Théorème 3.18 (Continuation)**.** *Pour toute exécution E et pour tout processus p qui invoque $adoptCommit(u)$ dans E, si un processus retourne avant que p n'invoque $adoptCommit$, p retourne $(adopt, v)$ ou $(commit, v)$, avec v une valeur proposée avant son invocation de $adoptCommit$.*

Démonstration. Soit E une exécution et p un processus qui invoque $adoptCommit(u)$ dans E. Supposons qu'un processus $q \neq p$ a retourné avant l'invocation de p.

q a soit écrit dans D (à la ligne 5 ou à la ligne 12), ou a lu une valeur qui n'est pas la valeur initiale dans D (à la ligne 4 ou à la ligne 10). Dans tous les cas, lorsque q retourne, D ne contient plus sa valeur initiale. Donc p ne peut pas retourner à la ligne 6. Et aucun processus ayant invoqué $adoptCommit$ après le retour de q ne peut exécuter les lignes 5 et 6.

Si p retourne à la ligne 4 ou à la ligne 10, alors il retourne la valeur lue dans D. Si cette valeur a été écrite par un processus ayant invoqué $adoptCommit$ avant l'invocation de $adoptCommit$ par p, alors la propriété est prouvée.

Supposons que la valeur a été écrite par un processus q' ayant invoqué $adoptCommit$ après l'invocation de $adoptCommit$ par p. Nous venons de prouver qu'aucun processus ayant invoqué $adoptCommit$ après le retour de q ne peut exécuter les lignes 5 et 6. Donc le processus q' a écrit dans D à la ligne 12. Donc q' a retourné à la ligne 13 ou à la ligne 14. Donc, d'après le Lemme 3.15, q' et q retournent la même valeur. Et donc p retourne une valeur proposée avant son invocation de $adoptCommit$ par q.

Si p retourne à la ligne 13 ou à la ligne 14, alors au moment de son exécution de la ligne 8, la variable C contient $false$. Donc le processus q qui a fini avant p n'a pas écrit dans C à la ligne 2 et n'a pas pu lire la valeur $true$ dans C. q a donc retourné à la ligne 13 ou à la ligne 14. Donc, d'après le Lemme 3.15, p et q retournent la même valeur. Et donc p retourne une valeur proposée avant son invocation de $adoptCommit$ par q. □

3.2.4 k-value-splitter versus k-converge

L'objet k-converge est une généralisation de l'adopt-commit, il a été présenté pour la première fois par Yang *et al.* [1998]. Il propose une opération $k-converge(u)$, avec u une valeur d'un ensemble V et qui retourne une paire $(flag, v)$, où $flag \in \{adopt, commit\}$ et le second élément v appartient à V. De plus, il satisfait les propriétés suivantes :

Validité La valeur de sortie d'une opération est la valeur d'entrée d'une opération (potentiellement d'une autre opération).

k-Accord Si un processus retourne $(commit, v)$, alors au plus k valeurs de sortie distinctes sont retournées.

k-Convergence S'il y a au plus k valeurs distinctes v telles qu'un processus a invoqué $k-converge(v)$, alors chaque processus p retourne $(commit, v_p)$, avec v_p l'une des k valeurs d'entrée.

La différence de propriété du 1-value-splitter avec l'adopt-commit (et le conflict-detector) est la propriété de convergence. De manière similaire, la propriété qui n'est pas assurée par le k-value-splitter (avec $k > 1$) est la propriété de k-convergence du k-converge. Dans le cas 1-value-splitter nous avons réussi, avec la même complexité (en pas de calcul et en espace), à proposer un conflict-detector. Dans ce cas où $k > 1$, nous proposons un algorithme de l'objet k-converge avec une complexité spatiale plus grande que le k-value-splitter, mais qui propose une amélioration de la complexité spatiale dans certain cas de l'algorithme proposé par Yang *et al.* [1998] (grossièrement lorsque $k < \sqrt(n)$).

Description L'algorithme du k-converge présenté dans l'Algorithme 14, fait appel à des conflict-detectors qui fournissent additionnellement la valeur qui a créé le conflit. Dans la suite, nous donnons des algorithmes de conflict-detectors invariant-aux-entrées (l'Algorithme 15) et non invariant-aux-entrées (l'Algorithme 16) qui fournissent la valeur additionnelle. Ils sont semblables respectivement aux algorithmes 8 et 11. Selon l'implémentation du conflict-detector, l'algorithme réalisant le k-converge est lui-même invariant-aux-entrées ou non.

Complexité Pour la version invariant-aux-entrées, la complexité en espace du k-converge est k fois la complexité en espace du conflict-detector plus $2 * (k-1)$. Ce qui donne une complexité de $O(k * \sqrt(n))$ avec l'implémentation du conflict-detector proposé.

Dans le cas non-invariant-aux-entrées, la complexité spatiale de chaque conflict-detector est différente. Soit m le nombre de valeurs d'entrée possibles du k-converge. Soit C_i la complexité spatiale du conflict-detector $CD[i]$. Pour tout $i = 0 \ldots, k-1$, $C_i = \frac{m!}{(i+1)!(m-i-1)!}$. Si $k < m/2$, alors pour tout $i = 0 \ldots, k-1$, $C_i \leq \frac{m!}{k!(m-k)!}$, sinon pour tout $i = 0 \ldots, k-1$, $C_i \leq \frac{m!}{m/2!m/2!}$.

```
Variables partagées :
Tableau de registres C[0...k−1], Initialement ∅
Tableau de registres W[0...k−1], Initialement ∅
Tableau de conflict-detectors CD[0...k−1]

Procédure : k-converge(v)
1  V ← {v};
2  for i = 0 to k − 1 do
3      < conflict, new_value > ← CD[i].check(V);
4      if i = k − 1 ∧ conflict = true then
5          Write(C[i], V ⋃{min(v'|v' ∈ new_value ∧ v' ∉ V)});
6          if ∃j|W[j] ≠ ∅ then
7              j ← max(j|W[j] ≠ ∅);
8              val ← Read(W[j]);
9              w ← min(v'|v' ∈ val);
10             return (adopt, w);
11         else
12             return (adopt, v);
13         end
14     else if conflict = false then
15         Write(W[i], V);
16         val ← Read(C[i]);
17         if val = ∅ then
18             return (commit, v);
19         else if i = k − 1 then
20             w ← min(v'|v' ∈ V);
21             return (adopt, w);
22         else
23             V ← V ⋃{min(v'|v' ∈ val ∧ v' ∉ V)};
24         end
25     else
26         Write(C[i], V ⋃{min(v'|v' ∈ new_value ∧ v' ∉ V)});
27         val ← Read(W[i]);
28         if val = ∅ ∨ val = V then
29             V ← V ⋃{min(v'|v' ∈ new_value ∧ v' ∉ V)};
30         else if v ∉ val then
31             V ← val ⋃{v};
32         else
33             V ← val ⋃{min(v'|v' ∈ V ∧ v' ∉ val)} ;
34         end
35     end
36 end
```

Algorithme 14: k-Converge

```
Variables partagées :
Tableau de registres R[0...b-1] avec b^2 - 4 > 4n et b ≥ 4. initialement ⊥
Procédure : check(v)
1  i ← 0;
2  next_writing ← 0;
3  while i < b - 1 do
4      i ← 0;
5      res ← Read(R[i]);
6      while i < b ∧ res ≠ ⊥ do
7          if res ≠ v then return true;
8          i++;
9          res ← Read(R[i]);
10     end
11     if i < b ∧ i = next_writing then
12         Write(R[i], v);
13         next_writing ← i + 1;
14     else
15         next_writing ← i;
16     end
17 end
18 return false;
```

Algorithme 15: Conflict-detector invariant-aux-entrées avec un retour de la valeur conflictuelle

```
Variables partagées :
Registres R[1..b], avec b! = m , où m est le nombre de valeurs d'entrée
distinctes possibles, initialement ⊥
Procédure : check(v)
1  for i := 1..k do
2      t := Read(R[π_v(i)]);
3      if t = ⊥ then Write(R[π_v(i)], v) if t ≠ v then return < true, t >
4  end
5  return < false, ⊥ >;
```

Algorithme 16: Conflict-detector non-invariant-aux-entrées avec un retour de la valeur conflictuelle

On pose $C_{max} = \frac{m!}{k!(m-k)!}$ si $k < m/2$ et $C_{max} = \frac{m!}{m/2!m/2!}$ sinon. Finalement, la complexité spatiale de l'implémentation du k-converge est en $O(k\frac{log(C_{max})}{log(log(C_{max}))})$.

Preuves Dans la suite, toutes les lignes auxquelles nous nous référons font référence à l'Algorithme 14. La propriété de validité est immédiate :

Théorème 3.19. ***Validité.*** *La valeur retournée est une valeur d'entrée d'une opération.*

De part la propriété d'accord du conflict-detector, nous avons l'observation suivante :

Observation 3.20. *Au plus une valeur est écrite dans $W[i]$ pour tout $i = 0 \ldots, k-1$.*

Lemme 3.21. *La valeur d'entrée pour toute opération check réalisée sur $CD[i]$ est un ensemble de cardinalité $i+1$ pour tout $i = 0, \ldots k-1$.*

Démonstration. Fixons une exécution de l'opération k-converge(v) par un processus p sur le k-converge. Au début de la i^{me} itération pour $i = 0$, l'ensemble V contient seulement $\{v\}$ (ligne 1). Par hypothèse d'induction, supposons que V a une cardinalité de $i+1$ au début de la i^{me} itération pour $i \geq 0$. Nous prouvons que V a une cardinalité de $i+2$ au début de la $i+1^{me}$ itération.

V est modifié aux lignes 23, 29, 31 et 33. Si V est modifié à la ligne 29, la propriété est satisfaite du fait que par hypothèse d'induction, V a une cardinalité de $i+1$ quand la ligne est exécutée et que new_value contient une valeur non présente dans V.

Supposons alors que V est modifié à la ligne 23. Nous prouvons que $V \bigcap val \neq \emptyset$, et donc par hypothèse d'induction la propriété est satisfaite. val contiens une valeur écrite dans $C[i]$ qui a été écrite par un processus q durant la i^{me} itération de l'exécution. D'après la ligne 26, q écrit dans $C[i]$ un ensemble $V_q \bigcup \{min(v) | v \in new_value \wedge v \notin V_q\}$ dont la cardinalité est $i+2$. En effet par hypothèse d'induction V_q a une cardinalité de $i+1$ et par définition du conflict-detector new_value contient une valeur non présente dans V_q.

Finalement, supposons que V est modifié à la ligne 31 ou à la ligne 33. Nous savons que $val \neq \emptyset$ et donc par l'hypothèse d'induction la valeur lue dans $W[i]$ est un ensemble de taille $i+1$. Si V est modifié à la ligne 31, nous savons également que $v \notin val$. Ainsi l'ensemble $V \cup \{v\}$ est de cardinalité $i+2$. De plus, nous savons que $V \neq val$. Donc, il existe au moins une valeur présente dans V qui ne l'est pas dans val. Et donc, à la ligne 33, la propriété est vérifiée par le fait du choix d'une valeur présente dans V qui ne l'est pas dans val. □

Théorème 3.22. ***k-convergence.*** *S'il y a au plus k valeurs distinctes v telles qu'un processus a invoqué k-converge(v), alors chaque processus p retourne $(commit, v_p)$, avec v_p l'une des k valeurs d'entrée.*

Démonstration. Un processus exécutant une opération k-converge retourne *adopt* seulement si l'opération check du conflict-detector $CD[k-1]$ retourne *true*. D'après le Lemme 3.21, toutes les valeurs d'entrée des opérations check réalisées sur $CD[k-1]$ sont un ensemble de taille k. Étant donné que l'ensemble des valeurs d'entrée du k-converge est au plus de k, aucun processus n'a un ensemble différent en entrée de $CD[k-1]$. Et donc, par la propriété du conflict-detector qui assure que si toutes les valeurs d'entrées sont les mêmes, toutes les sortie sont $false$, aucun processus retourne *true* lors de l'opération check sur le conflict-detector $CD[k-1]$. □

Observation 3.23. *L'ensemble $V_{i,p}$ que contient la variable V d'un processus p à l'itération $i > 0$ est un sur-ensemble de $V_{j,q}$, où $V_{j,q}$ est l'ensemble que contient la variable V d'un processus q à l'itération $j < i$.*

Lemme 3.24. *Si un processus p lors de sa i^{me} itération retourne $(commit, v)$, après avoir écrit V_i dans $W[i]$, avec $i < k-1$, alors, si un processus écrit V_j dans $W[j]$ avec $j > i$, $V_i \subset V_j$.*

Démonstration. Supposons qu'un processus p lors de sa i^{me} itération retourne $(commit, v)$, après avoir écrit V_i dans $W[i]$, avec $i < k-1$. Étant donné que p retourne $(commit, v)$, il a lu la valeur initiale dans $C[i]$. Soit un processus q qui va à l'itération $i+1$. S'il a modifié sa variable V à la ligne 23, d'après l'Observation 3.20, l'ensemble V_i est contenu dans l'ensemble que contient V. S'il a modifié sa variable V à la ligne 29, 31 ou 33, étant donné que p a lu la valeur initiale dans $C[i]$, q écrit dans $C[i]$ après que p ait écrit dans $W[i]$, et donc q lit la valeur V_i dans $W[i]$. Et donc, l'ensemble V_i est contenu dans l'ensemble que contient V.

Donc, tous les processus qui ne retournent pas à l'itération i, ont modifié leur variable V de manière à ce qu'elle contienne un sur-ensemble de V_i. Étant donné qu'un processus écrit dans W le contenu de sa variable V, et d'après l'Observation 3.23, la propriété est prouvée. □

Observation 3.25. *La valeur d'entrée v de l'opération k-converge d'un processus est toujours contenue dans l'ensemble contenu dans la variable locale V du processus.*

Théorème 3.26. ***k-Accord.*** *Si un processus retourne $(commit, v)$, alors au plus k valeurs de sortie distinctes sont retournées.*

Démonstration. Supposons i l'itération maximale à laquelle au moins un processus a retourné *commit*, avec $i = 0 \ldots, k-1$. Soit V_i l'ensemble écrit dans $W[i]$. Si un processus a retourné *commit* lors d'une itération antérieure, alors la valeur retournée est contenue dans V, d'après le Lemme 3.24 et l'Observation 3.25. De plus, d'après le Lemme 3.21, $|V_i| = i+1$.

Nous allons prouver que les valeurs renvoyées par les processus renvoyant *adopt* appartiennent soit à V_i, soit à un ensemble V' tel que $|V'| \leq k-i-1$. Pour

qu'un processus renvoie *commit* à l'itération i, il doit lire la valeur initiale dans $C[i]$, donc $W[i]$ est écrit avant que $C[i]$ le soit. Donc, si un processus renvoie *adopt*, il le fera à la ligne 21 ou à la ligne 10. S'il retourne à la ligne 21, il retourne la valeur minimale de l'ensemble écrit dans $W[k-1]$. De plus, s'il retourne à la ligne 10, il lira $W[i] \neq \emptyset$, donc retournera la valeur minimale de l'ensemble écrit dans $W[i]$, c'est-à-dire une valeur de V_i ou la valeur minimale de l'ensemble d'un registre $W[j]$ avec $j > i$. Soit V' l'ensemble des valeurs minimales des ensembles écrit dans $W[j]$, avec $k-1 \geq j > i$.

Finalement, les processus renvoient soit une valeur de V_i avec $|V_i| = i+1$, soit une valeur de V' avec $|V'| \leq k-i-1$. □

3.3 Résumé

Dans ce chapitre, nous avons présenté une nouvelle abstraction pour résoudre le k-accord : le k-value-splitter. Deux implémentations anonymes de cette abstraction sont proposées. La première est invariant-aux-entrées et a une complexité en espace de $O(\sqrt{n/k})$, quelque soit le nombre m d'entrées possibles. La seconde a une complexité de $O(\log(m/k)/\log\log(m/k))$, quelque soit le nombre de processus n.

Dans la suite, cette abstraction nous a permis de faire le lien entre les autres abstractions existantes dans l'état-de l'art, permettant de mettre en évidence les différences entre les propriétés des différentes abstractions. Il est ainsi possible de comprendre quelles peuvent être les propriétés responsables d'un surcoût en terme de complexité spatiale. Nous avons également proposer des améliorations d'implémentation en terme de complexité en espace du conflict-detector et du k-converge.

Chapitre 4

Consensus et k-accord

Le consensus et sa généralisation, le k-accord, sont des objets centraux dans le calcul distribué, comme nous l'avons vu dans le Chapitre 1. Nous nous sommes intéressés à savoir quelle pouvait être la complexité spatiale de ces objets avec les propriétés d'intervalle-solo-rapidité ou d'abandon, peu étudiés par rapport à leur variante sans obstruction.

Ce chapitre est donc consacré aux travaux sur le consensus et le k-accord ayant, soit la propriété d'intervalle-solo-rapidité, soit étant abandonnable. Nos résultats, concernant le k-accord avec la propriété d'intervalle-solo-rapidité (donc sans attente par définition), sont présentés dans la première partie de ce chapitre. Nous avons présenté dans Capdevielle *et al.* [2015] une borne inférieure sur le nombre de registres nécessaires pour le consensus ainsi qu'un algorithme correspondant en complexité à cette borne pour un consensus intervalle-solo-rapide. Nous présentons ici une généralisation de ces résultats pour le k-accord. Un schéma récapitulatif de cette partie est présent à la Figure 4.1.

La seconde partie de ce chapitre concerne les résultats sur le consensus sans attente avec une propriété d'abandon. Nous donnons également un schéma récapitulatif de cette partie à la Figure 4.4.

4.1 k-accord intervalle-solo-rapide

Dans cette partie, nous nous intéressons à l'implémentation d'un k-accord intervalle-solo-rapide. Pour rappel, l'utilisation de primitives CAS est limitée. Plus précisément, l'algorithme qui implémente une opération utilise des primitives CAS seulement si cette opération s'exécute concurremment.

Pour un algorithme non anonyme, c'est à dire que l'on peut utiliser les identifiants des processus dans l'algorithme, l'algorithme de consensus présenté par Luchangco *et al.* [2003] donne directement une complexité spatiale de $\theta(1)$. Ainsi, les premiers résultats que nous présentons se font pour les algorithmes anonymes, c'est-à-dire que les identifiants des processus ne sont pas utilisés.

Un résumé est donné à la Figure 4.1. Dans cette figure, la flèche entre la propriété step-solo-rapide et intervalle-solo-rapide symbolise le fait que la propriété de step-solo-rapidité assure par définition celle d'intervalle-solo-rapidité. Ainsi les bornes inférieures sur la complexité spatiale qui existent sur le consensus intervalle-solo-rapide sont également valables pour le consensus step-solo-rapide. À l'inverse la borne supérieur ne peut pas s'appliquer automatiquement. Nous rappelons également le résultat de Attiya *et al.* [2009] qui donne une borne supérieur de $O(n)$ pour un consensus non anonyme step-solo-rapide.

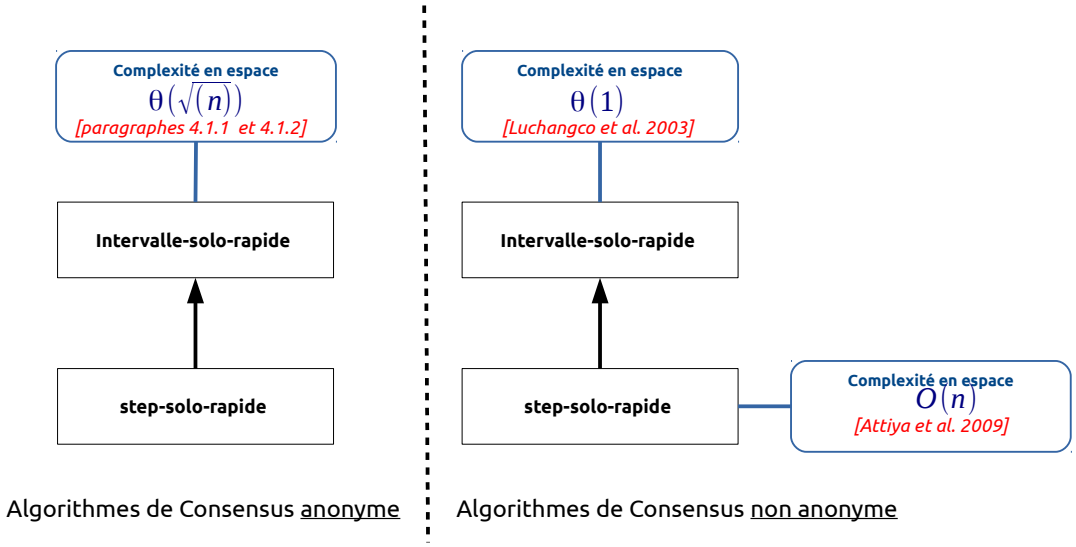

FIGURE 4.1 – Résultats sur la complexité spatiale des Consensus avec propriétés de solo-rapidité

4.1.1 Borne inférieure

Cette section est dédiée à la présentation de la borne inférieure de $\Omega(\min(\sqrt{n/k}, \log(m/k)/\log\log(m/k)))$ sur le nombre de registres nécessaires dans un système anonyme pour l'implémentation d'un k-accord. Dans le cas où l'algorithme est additionnellement *invariant-aux-entrées*, la borne inférieure devient $\Omega(\sqrt{n/k})$. Nous nous intéressons ici à une exécution où les processus s'exécutent du début jusqu'à la fin sans observer aucune concurrence.

Cette borne est valable pour les objets k-accord solo-rapide, mais également pour ceux abandonnables et sans obstruction (notons que de meilleures bornes existent pour ces derniers).

Aperçu de la preuve

Par contradiction, supposons qu'il existe un algorithme A d'un k-accord intervalle-solo-rapide, tel que au plus b objets de base distincts soient écrits

dans les exécutions solo de A et que $b < \min(\sqrt{2n/(k+1)}, \Gamma^{-1}(m/k))$. Γ^{-1} étant l'inverse de la fonction factorielle $\Gamma(x) = x!$, rappelons que $\Gamma^{-1}(x) \in \Theta(\log x / \log\log x)$. Nous allons établir que l'algorithme a une exécution dans laquelle $k+1$ différentes valeurs sont retournées.

Pour atteindre ce but, nous construisons de manière itérative une exécution indistinguable pour tous les processus d'une exécution dans laquelle chacun d'entre eux s'exécute seul. Aucun processus n'effectuera donc des primitives autres que des primitives *Read* et *Write*. Dans cette exécution, $k+1$ processus s'exécutent jusqu'à ce que leur opération soit complète, écrivant sur tous les objets de base, et décidant leur valeur d'entrée. Les autres processus seront utilisés pour cacher la concurrence : les processus qui décident ne doivent pas pouvoir lire une autre valeur que celle qu'ils ont écrit dans les objets de base.

Soit C_0 la configuration initiale de A. Pour chaque valeur $u \in V$, définissons α_u comme l'exécution de A dans laquelle un processus, partant de C_0, invoque *propose*(u) et s'exécute seul jusqu'à ce que l'opération termine. Étant donné que l'algorithme est anonyme, α_v ne dépend pas de l'identifiant du processus.

Pour un $u \in V$ donné, considérons la séquence des objets de base écrits dans α_u, ordonnés par le temps de la *première écriture* dans α_u. Pour la preuve, il faut, pour $k+1$ différentes valeurs (u_0, ..., u_k), la même séquence d'objets de base ordonnés par le temps de la *première écriture* dans α_{u_i}.

Dans le cas d'un algorithme invariant-aux-entrées (*i.e.* un processus accède à la même séquence d'objets de base lors d'une exécution solo quelque soit l'entrée), toutes ces séquences sont identiques, indépendamment de m et b. Donc l'ensemble $\{u_o, \ldots, u_k\}$ peut être n'importe quel ensemble de $k+1$ valeurs d'entrée distinctes.

Si ce n'est pas le cas, cet ensemble existe du fait de l'hypothèse $b < \Gamma^{-1}(m/k)$. En effet, il y a m possibles valeurs d'entrée u (et donc autant d'exécutions α_u), et au plus $b!$ ordres possibles des premières écritures des objets de base dans l'exécution α_u, $u \in V$. Comme $b < \Gamma^{-1}(m/k)$, nous avons $b! < m/k$ et donc il existe $k+1$ valeurs distinctes $\{u_0, \ldots, u_k\}$ telles que les séquences des objets de base, ordonnés selon le temps où ils ont été pour la première fois écrits lors de l'exécution de α_{u_i} pour $i = 1, .., k$, sont identiques.

Notons cette séquence d'objets de base $r_1, \ldots, r_b$ et fixons la pour le reste de la preuve. Nous construisons l'exécution désirée en b itérations telle que à la $i$ème itération, le $i$ème objet de base dans $r_1, \ldots, r_b$ soit écrit pour la première fois et tous les processus qui continuent à faire des pas ne peuvent pas distinguer cette exécution d'une autre exécution dans laquelle ils seraient seuls.

L'ensemble des processus est divisé en $k+1$ groupes, de telle sorte que les processus du groupe i proposent la valeur u_i, avec $i = 0, \ldots, k$. A chaque itération de la construction, on "réveille" un sous-ensemble des processus de chaque groupe et on les laisse s'exécuter en tant que *clones* (*i.e.* ils s'exécutent de manière lock-step, lisant et écrivant les même valeurs, les uns après les autres,

restant tous dans le même état, de sorte qu'ils ignorent la présence des autres groupes), jusqu'à ce qu'ils soient sur le point d'écrire sur un nouvel objet de base pour la première fois. Dans le même temps, un invariant sur le fait que chaque objet de base est *couvert* par "suffisamment" de processus de chacun des $k+1$ groupes est maintenu : un processus p *couvre un objet de base* r *à une configuration donnée* C si p est sur le point d'écrire dans r à C. Intuitivement, ces "couvertures" par des primitives d'écriture, une fois effectuées, assurent qu'un groupe de processus ne s'apercevra pas de la présence d'un autre groupe lors du prolongement de l'exécution. Il en résulte que "suffisamment" de processus de chacun des $k+1$ groupes ne distinguent pas l'exécution construite d'une exécution solo.

L'hypothèse que $b < \sqrt{2n/(k+1)}$ assure qu'il y a suffisamment de processus pour construire l'exécution de telle sorte qu'il y ait à la fin de la b-ième itération, pour chaque $i = 0, \ldots, k$, au moins un processus q_i du groupe i qui ne peut pas distinguer l'exécution actuelle d'une exécution où il s'exécute seul. De plus, à la configuration résultante C_b, chacun des objets de base $r_1, \ldots, r_b$ est couvert par des processus prêts à écrire la dernière valeur écrite par q_i. Ainsi, pour chaque $i = 0, \ldots, k$, successivement, l'exécution est prolongée en laissant les processus couvrir les objets de base pour qu'ensuite le processus q_i s'exécute jusqu'à ce qu'il décide. De cette manière, $k+1$ valeurs distinctes sont décidées, ce qui établit la contradiction.

Par conséquent, le nombre de registres utilisés par un tel algorithme est au moins de $\min(\sqrt{2n/(k+1)}, \Gamma^{-1}(m/k))$, ce qui donne asymptotiquement $\Omega(\min(\sqrt{n/k}, \Gamma^{-1}(m/k)))$. Dans le cas d'un algorithme invariant-aux-entrées, l'hypothèse $b < \Gamma^{-1}(m/k)$ est inutile, la borne est donc $\Omega(\sqrt{n/k})$.

Notations et définitions

Avant de passer à la preuve de la borne inférieure, voici un ensemble de notations et de définitions utilisées.

Pour rappel, α_u désigne l'exécution complète et solo de *propose*(u) à partir de la configuration initiale C_0. Pour $u \in \{u_0, \ldots, u_k\}$, $1 \leq i \leq b$, soit $\alpha_{i,u}$ le plus long préfixe de α_u qui contient seulement des écritures sur des objets de base appartenant à $\{r_1, \ldots, r_i\}$. Soit $\alpha_{0,u}$ le plus long préfixe de α_u dans lequel aucune écriture a eu lieu. Par définition, $\alpha_{b,u} = \alpha_u$, et pour tous $0 \leq i \leq b-1$, le prochain événement de α_u immédiatement après $\alpha_{i,u}$ est une écriture dans r_{i+1}.

A noter que chaque objet de base peut être écrit plusieurs fois pendant l'exécution de α_u. Pour $j = 1, \ldots, b$, notons $x_{j,i,u}$ la valeur de r_j dans la configuration juste après $\alpha_{i,u}$. Par définition de $\alpha_{i,u}$, pour $j = i+1, \ldots, b$, aucune écriture ne se déroule dans r_j pendant $\alpha_{i,u}$ et donc $x_{j,i,u}$ est la valeur initiale de r_j.

Pour $i, j = 1, \ldots, b$ et $u \in \{u_0, \ldots, u_k\}$, $I_{j,i,u}$ est un indicateur binaire du

fait que r_j soit écrit dans $\alpha_{i,u}$ après le dernier événement de $\alpha_{i-1,u}$. Ainsi, $I_{i,i,u} = 1$ pour $i = 1, \ldots, b$, et $I_{j,i,u} = 0$ pour $1 \leq i < j \leq b$. Pour illustrer notre propos, considérons l'exécution solo d'une opération $propose(u)$ représentée Figure 4.2. Ici, $I_{1,2,u}$ est l'indicateur binaire que r_1 est écrit pendant $\alpha_{2,u}$ après le dernier événement de $\alpha_{1,u}$, *i.e.* dans le fragment d'exécution δ. S'il y a une écriture de r_1 pendant δ, alors $I_{1,2,u} = 1$, sinon $I_{1,2,u} = 0$.

Pour cacher la concurrence, il faut assurer que suffisamment de processus de chaque groupe soient sur le point d'écrire la dernière valeur écrite dans les objets de base précédemment écrits. La dernière valeur écrite dépend du préfixe $\alpha_{i,u}$ de α_u déjà émulé. Le nombre de processus correspondant pour un objet de base donné dépend de quand cet objet de base va être à nouveau écrit dans l'extension de $\alpha_{i,u}$.

Formellement, pour $1 \leq j \leq i < b$, nous définissons $s_{j,i,u}$ comme 1 plus le nombre maximal de préfixes consécutifs $\alpha_{t,u}$, tel que $i < t < b$ et r_j ne soit pas écrit dans $\alpha_{t,u}$ après le dernier événement de $\alpha_{t-1,u}$, *i.e.* $s_{j,i,u} = \min\{\ell > i | \ell = b \vee I_{j,\ell,u} = 1\} - i$.

La Figure 4.3 représente un fragment d'une exécution α_u et explique le notation $s_{j,i,u}$ graphiquement. En particulier, pour un objet de base donné r_j et un préfixe donné $\alpha_{i,u}$, nous considérons la plus longue séquence de fragment d'exécution, tel que chaque fragment soit distinct et consécutif. Partant de $t = i + 1$ chaque fragment débute à la première écriture de r_t et s'arrête à la première écriture (mais sans l'inclure) dans r_{t+1}. Enfin ces fragments ne contiennent aucune écriture de r_j. Cette séquence de fragments est notée par β dans l'illustration. Ainsi, $s_{j,i,u}$ est simplement le nombre de fragments consécutifs dans β plus un, c'est-à-dire $\ell - i$ dans ce cas, étant donné que le fragment γ entre la première écriture de r_ℓ jusqu'à la première écriture dans $r_{\ell+1}$ contient une écriture sur r_j.

Clairement, $s_{j,i,u} \geq 1$ et $s_{j,b-1,u} = 1$, pour tous $i, j = 1, \ldots, b-1$. Il est simple de vérifier également que, $\sum_{i=1}^{b-1} I_{j,i,u} s_{j,i,u} = b - j$ pour tous $j = 1, \ldots, b-1$. Donc, $\sum_{j=1}^{b-1} \sum_{i=1}^{b-1} I_{j,i,u} s_{j,i,u} = (b^2 - b)/2$.

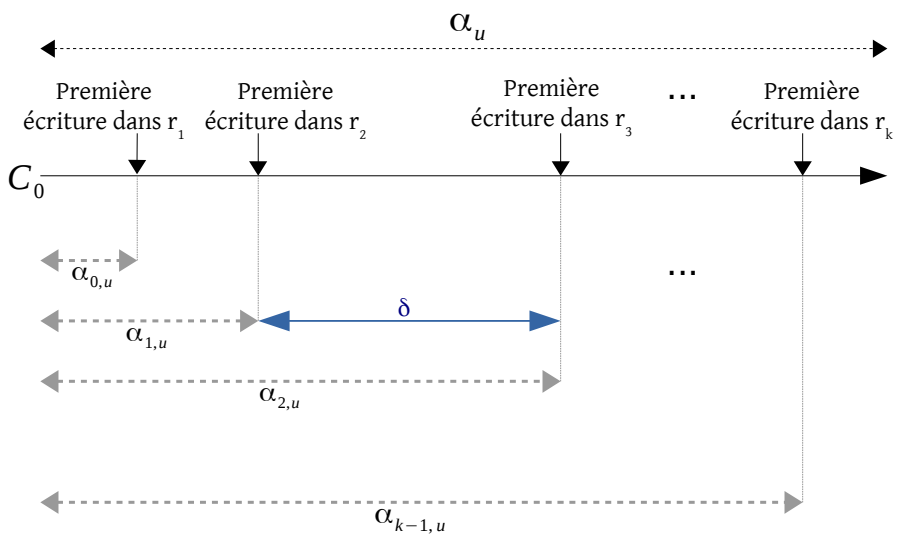

FIGURE 4.2 – Exécution solo de $propose(u)$, notée α_u, par un processus p.

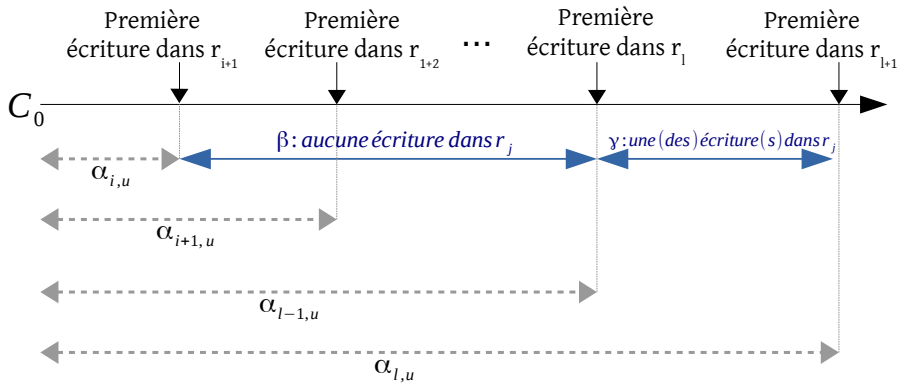

FIGURE 4.3 – Définition de $s_{j,i,u}$: β contient $\ell - i$ fragments consécutifs dans lesquels r_j n'est pas écrit.

Configuration de clonage

Nous introduisons maintenant la notion centrale de la borne inférieure :

Définition 4.1. Une configuration C_i, $i = 1, \ldots, b$, est appelée $(i, k+1)$-*clonage* si :

— Pour tous $u \in \{u_0, \ldots, u_k\}$, $j = 1, \ldots, i-1$, r_j est couvert par $s_{j,i-1,u}$ processus écrivant $x_{j,i-1,u}$.

— Pour chaque $u \in \{u_0, \ldots, u_k\}$, il y a au moins $(b^2-b)/2+1-\sum_{j=1}^{i-1}\sum_{\ell=1}^{i-1} I_{j,\ell,u}s_{j,\ell,u}$ pour $i > 1$ et $(b^2-b)/2+1$ pour $i = 1$ processus qui ne distinguent pas l'exécution de $\alpha_{i-1,u}$ et donc couvre un objet de base r_i avec la valeur $x_{i,i,u}$.

— Chaque objet de base dans $\{r_i, \ldots, r_b\}$ contient la valeur initiale.

Lemme 4.1. *Soit A un algorithme anonyme de k-accord intervalle-solo-rapide pour n processus et m valeurs. Si au plus b objets de base distincts sont écrits dans toute exécution solo de A, où (1) $b < \sqrt{2n/(k+1)}$ et où (2) soit A est invariant-aux-entrées soit $b < \Gamma^{-1}(m/k)$), alors A possède une configuration $(b, k+1)$-clonage.*

Démonstration. La construction de la configuration $(b, k+1)$-clonage, se fait par induction, partant de la configuration initiale C_0 de A.

Les processus sont divisés en $k+1$ groupes de taille d'au moins $1 + (b^2 - b)/2$ processus ; chacun des processus du $i$ème groupe, avec $i = \{0, \ldots, k\}$, proposant la valeur u_i. Comme $b < \sqrt{2n/(k+1)}$, ce découpage est possible et donc, en supposant que $b \geq 2$, $((b^2 - b)/2 + 1)(k+1) \leq n$.

Cas de base. Soit γ la concaténation de toutes les exécutions de A dans lesquelles, commençant de C_0, un processus distinct s'exécute en isolation jusqu'à qu'il soit sur le point d'écrire sur l'objet de base r_1 pour la première fois. Sachant que r_1 est le premier objet de base écrit dans tous α_{u_i}, aucun processus ne peut distinguer γ de son exécution solo et donc γ est une exécution de A.

Comme aucun processus n'a encore écrit dans γ, $C_1 = C_0\gamma$ est une configuration $(1, k+1)$-clonage. En effet, aucun processus dans un groupe i, $i = 0, \ldots, k$, peut distinguer $C_0\gamma$ de l'exécution α_{0,u_i}, et tous les objets de base sont dans leurs états initiaux.

Par *hypothèse d'induction*, supposons que pour un $1 \leq i < b$, la configuration C_i est une configuration $(i, k+1)$-clonage. Pour chaque $u \in \{u_0, \ldots, u_k\}$, la procédure suivante est effectuée.

Premièrement, pour tous $j = 1, \ldots, i-1$, un des processus couvrant r_j avec $x_{j,i-1,u}$ fait son écriture. Par l'hypothèse d'induction, il y a au moins $s_{j,i-1,u} \geq 1$ processus de ce type. À noter que dans le cas où $i = 1$, il n'y pas de tel processus et l'exécution est laissée inchangée.

Ensuite, les $1 + (b^2 - b)/2 - \sum_{j=1}^{i-1} \sum_{\ell=1}^{i-1} I_{j,\ell,u} s_{j,\ell,u}$ processus qui ne peuvent distinguer l'exécution de $\alpha_{i-1,u}$ s'exécutent de manière *lock-step*, jusqu'à ce qu'ils soient sur le point d'effectuer leur première écriture sur r_{i+1}. Aucun d'entre eux ne peut distinguer l'exécution de $\alpha_{i,u}$, et donc nous obtenons une exécution de A. Si $\alpha_{i,u}$ contient une écriture sur des objets r_j, $j = 1, \ldots, i$, après le dernier événement de $\alpha_{i-1,u}$, alors $s_{j,i,u}$ de ses processus s'arrêtent juste avant d'effectuer la dernière écriture sur r_j de $\alpha_{i,u}$. Cela peut être fait car le nombre total de ces processus est $\sum_{j=1}^{i} I_{j,i,u} s_{j,i,u}$, et dans C_i pour chaque $u \in \{u_0, \ldots, u_k\}$, il y en a au moins $(b^2 - b)/2 + 1 - \sum_{j=1}^{i-1} \sum_{\ell=1}^{i-1} I_{j,\ell,u} s_{j,\ell,u}$. Et $1 + (b^2 - b)/2 = 1 + \sum_{j=1}^{b-1} \sum_{\ell=1}^{b-1} I_{j,\ell,u} s_{j,\ell,u} > \sum_{j=1}^{i} \sum_{\ell=1}^{i} I_{j,\ell,u} s_{j,\ell,u}$.

Soit γ l'extension résultante de C_i et $C_{i+1} = C_i\gamma$ la configuration résultante. Notons que dans C_{i+1}, tous les objets de base de $\{r_{i+1}, \ldots, r_b\}$ ont toujours leur états initiaux.

Pour tous $j = 1, \ldots, i$ et $u \in \{u_0, \ldots, u_k\}$, si r_j n'est pas écrit pendant $\alpha_{i,u}$ alors par l'hypothèse d'induction et la construction de γ, r_j est couvert par

$s_{j,i,u} = s_{j,i-1,u} - 1$ processus écrivant $x_{j,i,u} = x_{j,i-1,u}$. Sinon, par construction r_j est couvert par $s_{j,i,u}$ processus écrivant $x_{j,i,u}$.

Finalement, pour chaque $u \in \{u_0, \ldots, u_k\}$, comme $\sum_{j=1}^{i} I_{j,i,u} s_{j,i,u}$ processus additionnels sont utilisés pour couvrir les objets de base $r_1, \ldots, r_i$, au moins $(b^2 - b + 2)/2 - \sum_{\ell=1}^{i} \sum_{j=1}^{\ell-1} I_{j,\ell,u} s_{j,\ell,u}$ processus restants ne peuvent distinguer $C_i\gamma$ de $C_0\alpha_i$ et donc ces processus doivent couvrir r_{i+1}.

Ainsi, C_{i+1} est une configuration $(i+1, k+1)$-clonage, et par induction A possède une configuration $(b, k+1)$-clonage. □

Théorème 4.2. *Tout algorithme A anonyme de k-accord intervalle-solo-rapide, pour n processus et m valeurs, doit avoir une complexité en espace de $\Omega(\min(\sqrt{n/k}, \log(m/k)/\log\log(m/k)))$. De plus, si l'algorithme est invariant-aux-entrées, la borne devient $\Omega(\sqrt{n/k})$.*

Démonstration. Supposons par contradiction qu'un algorithme anonyme de k-accord intervalle-solo-rapide pour n processus et m valeurs utilise b objets de base tel que $n \geq (b^2-b+2)(k+1)/2$ et que soit $b < \Gamma^{-1}(m/n)$, soit l'algorithme est invariant-aux-entrées.

D'après le Lemme 4.1, il existe une configuration C_b $(b, k+1)$-clonage pour des valeurs d'entrée $\{u_0, \ldots, u_k\}$. A la configuration C_b, pour tout $u \in \{u_0, \ldots, u_k\}$, chaque objet de base r_j, $j = 1, \ldots, b-1$, est couvert par exactement $s_{j,b-1,u} = 1$ processus écrivant la valeur $x_{j,b-1,u}$. Il y a également au moins $(b^2-b)/2+1-\sum_{\ell=1}^{b-1}\sum_{j=1}^{b-1} I_{j,\ell,u} s_{j,\ell,u} = (b^2-b)/2+1-\sum_{j=1}^{b-1}(b-j) = 1$ processus qui ne peuvent distinguer l'exécution de $\alpha_{b-1,u}$ et donc doivent couvrir r_b avec $x_{b,b,u}$.

Pour chaque $u \in \{u_0, \ldots, u_k\}$ successivement, laissons les processus couvrant r_j, $j = 1, \ldots, b-1$ avec la valeur $x_{j,b-1,u}$ effectuer leur écriture, puis le processus proposant u et couvrant r_b s'exécuter. Le processus ne distinguant pas cette exécution de $\alpha_{b,u}$ il terminera finalement en retournant la valeur u.

Ainsi $k+1$ valeurs différentes sont retournées dans l'exécution résultante, impliquant une contradiction.

Donc comme $\Gamma^{-1}(m/k) = \Theta(\log(m/k)/\log\log(m/k))$, l'algorithme a une exécution solo dans laquelle $\Omega(\min(\sqrt{n/k}, \log(m/k)/\log\log(m/k)))$ registres distincts sont nécessaires. De plus, si l'algorithme est invariant-aux-entrées, la borne inférieure devient $\Omega(\sqrt{n/k})$.

□

Remarque : Le Lemme 4.1 montre que d'avoir au moins $(k+1)(b^2-b+2)/2$ processus est suffisant pour construire une configuration $(b, k+1)$-clonage et donc d'établir une contradiction. La borne inférieure peut être raffinée à $k(b^2 - b + 2)/2 + 1$, si on organise l'exécution de chaque itération de la construction, inductive de C_b.

A chaque itération i, pour $i = 1, \ldots b$ les processus s'exécutant en premier sont ceux ayant été les derniers à écrire à l'itération $i-1$. Par ce moyen,

chaque objet de base r_j, $j = 1, \ldots, i-1$ stocke la dernière valeur qu'on écrit ces processus, qui est $x_{j,i-1,w}$. Ce qui réduit de $i-1$ les processus utilisés pour couvrir les objets de base $r_1, \ldots r_{i-1}$ avec la valeur $x_{j,i-1,w}$. Donc, cela permet une économie de $(b^2 - b)/2$ processus couvrants. Donc, au total, le nombre nécessaire de processus est $k(b^2 - b + 2)/2 + 1$, ce qui fait un nombre b plus proche de la borne supérieure $k(b^2 - 3b + 4)/2 + 1$, présentée dans la section suivante. Pour la simplicité de la présentation, la borne inférieure plus grossière (mais asymptotiquement équivalente) a été choisie.

4.1.2 k-accord intervalle-solo-rapide optimal

Dans cette section, nous présentons l'algorithme qui implémente un k-accord intervalle-solo-rapide. Cet algorithme peut être vu comme la généralisation de l'algorithme du consensus basé sur un *splitter* de Luchangco *et al.* [2003], où l'on remplace l'objet *splitter* par un objet *k-value-splitter* qui a été introduit dans le chapitre précédent. La propriété d'intervalle-solo-rapidité du k-accord est assuré si le k-value-splitter est lui-même intervalle-solo-rapide. De plus, en terme de complexités (en pas de calcul et en espace), l'algorithme a un surcoût constant par rapport au coût de l'implémentation du k-value-splitter utilisé.

k-accord utilisant un k-value-splitter

Le pseudo-code de l'algorithme de k-accord est donné dans l'Algorithme 17. Une valeur décidée par le k-accord est écrite dans la variable D, initialisée à $\perp \notin V$. Le premier pas que fait un processus est de vérifier si D stocke une valeur différente de la valeur initiale $\perp$, et si c'est le cas, il retourne cette valeur. Dans le cas contraire, le processus accède à l'objet k-value-splitter kVS pour vérifier s'il y a une autre opération *propose* qui s'exécute en même temps.

Si un processus p obtient *true* lors de son invocation de $kVS.split(v)$, il écrit sa valeur d'entrée v dans le registre F. Alors, il lit le registre Z pour vérifier la présence de processus ayant détecté une concurrence, et si la valeur est à $false$ (*i.e.* pas de concurrence), le processus p décide sa propre valeur. Avant de retourner la valeur décidée, il l'écrit dans le registre D.

Les écritures dans F et D, ainsi que la lecture de Z entre les deux, servent à assurer que soit le processus p détecte qu'un autre processus s'exécute et a recourt à l'utilisation d'une primitives CAS, soit le processus en concurrence adopte la valeur d'entrée de p.

Si p obtient $false$ du k-value-splitter, il met Z à vrai (*i.e.* une concurrence est détectée). Pour rappel, cela peut se passer si plus d'un processus accèdent au k-value-splitter, quelles que soient les valeurs d'entrée. Alors, p lit le registre F, et si F stocke une valeur différente de $\perp$, il adopte cette dernière comme étant sa propre proposition. Enfin, il effectue la primitive CAS sur D avec sa

proposition et décide la valeur lue dans D.

À noter qu'en supposant que le k-value-splitter est intervalle-solo-fast, un processus qui s'exécute en l'absence de concurrence atteint la décision en effectuant seulement des primitives $Read$ et $Write$.

Dans la suite, nous prouvons que l'Algorithme 17 implémente effectivement un k-accord intervalle-solo-rapide. La preuve est conditionnée par le fait que kVS est une implémentation d'un k-value-splitter intervalle-solo-rapide.

```
   Variables partagées :
   D, F, initialement ⊥
   Z, initialement false
   k-value-splitter kVS
   Procédure : propose(v)
 1 if (t := Read(D)) ≠ ⊥ then return t
 2 if kVS.split(v) then
 3 |   Write(F, v);
 4 |   if ¬(Read(Z)) then
 5 |   |   Write(D, v);
 6 |   |   return v
 7 |   end
 8 else
 9 |   Write(Z, true);
10 |   if (t := Read(F)) ≠ ⊥ then v := t;
11 end
12 CAS(D, ⊥, v);
13 res := Read(D);
14 return res
```

Algorithme 17: k-accord intervalle-solo-rapide

Preuves de l'algorithme du k-accord intervalle-solo-rapide (Algorithme 17)

Lemme 4.3 (Accord)**.** *Pas plus de k différentes valeurs sont retournées.*

Démonstration. Étant donné que seules les valeurs écrites dans D sont retournées, il suffit de prouver qu'au plus k valeurs peuvent être écrites dans D.

La variable D est mis à jour aux lignes 12 et 5. Comme le CAS permet d'écrire une valeur dans D à la ligne 12 seulement si D contient $\perp$, et comme D est écrit avec une valeur différente de $\perp$, au plus un processus peut réussir à écrire avec la primitive CAS. D est écrit à la ligne 5 seulement si le processus correspondant obtient *true* du k-value-splitter. Par la propriété de k-VS-accord du k-value-splitter, au plus k valeurs distinctes peuvent être écrites dans D à la ligne 5.

Donc, la seule possibilité pour que $k+1$ valeurs soient écrites dans D est qu'un processus, nommé p, effectue un CAS ligne 12 et met à jour D avec une valeur v, puis que d'autres processus écrivent k valeurs distinctes dans D à la ligne 5. Supposons par contradiction que cela soit vrai. Par la propriété de k-VS-accord du k-value-splitter, d'une part au plus k valeurs différentes de $\perp$ peuvent être lues dans F. D'autre part, le processus p doit obtenir $false$ du k-value-splitter, sinon seulement $k-1$ valeurs autres que celles de p peuvent être écrites dans D ligne 5. Donc, avant d'effectuer le CAS sur D, p a lu F à la ligne 10.

Soit p a lu une valeur différente de $\perp$ dans la variable F et donc il a effectué son CAS après avoir adopté la valeur de F (ligne 10). Il écrit donc une valeur dans D qui va être écrite par un des autres processus écrivant dans D à la ligne 5. Ainsi, seulement k valeurs distinctes sont écrites, ce qui contredit l'hypothèse faite.

Soit p lit F avant qu'aucun autre processus n'écrivent dans F. En regardant le pseudo-code de l'algorithme, on voit que p a écrit préalablement dans Z la valeur $true$ à la ligne 9. Par conséquent, après n'importe quelle écriture dans F, un processus lit la valeur $true$ dans Z (*i.e.* concurrence détectée) et tente un CAS au lieu d'écrire dans D ligne 5. Ainsi, aucune valeur n'est écrite dans D à la ligne 5, ce qui contredit également l'hypothèse faite. □

Lemme 4.4. *L'algorithme est intervalle-solo-rapide.*

Démonstration. Si un processus p invoque son opération *propose* et trouve une valeur différente de $\perp$ dans D, alors p retourne après avoir effectué une seule primitive *read* sur D, la propriété est prouvée.

Supposons maintenant que p lit $D = \perp$ et effectue une primitive CAS (à la ligne 12). Nous allons prouver qu'il a une opération qui intersecte la proposition *propose* de p.

Par l'inspection du pseudo-code, il est aisé de voir que p effectue une primitive CAS seulement si (1) il a lu $Z = true$ (à la ligne 4) ou (2) il a obtenu $false$ du kVS. Dans les deux cas, par la propriété d'exécution solo du k-value-splitter, il y a un autre processus q qui a invoqué $kVS.split(v)$ avant que p n'ai fini son opération *propose*.

Avant de compléter son opération, q a écrit sa valeur décidée dans D. Étant donné que p a lu dans D la valeur initiale, l'opération de q n'est pas finie avant que l'opération de p commence.

Donc deux opérations s'intersectent. En supposant que le k-value-splitter est intervalle-solo-rapide et par le fait que l'algorithme ne contient ni boucle ni état d'attente, la propriété est prouvée.

□

Les Lemmes 4.3 et 4.4 implique :

Théorème 4.5. *Si kVS est une implémentation d'un k-value-splitter intervalle-solo-rapide, alors l'Algorithme 17 implémente un k-accord intervalle-solo-rapide avec une complexité en espace de $O(b)$, avec b la complexité en espace du kVS*

La complexité provient directement du pseudo-code.

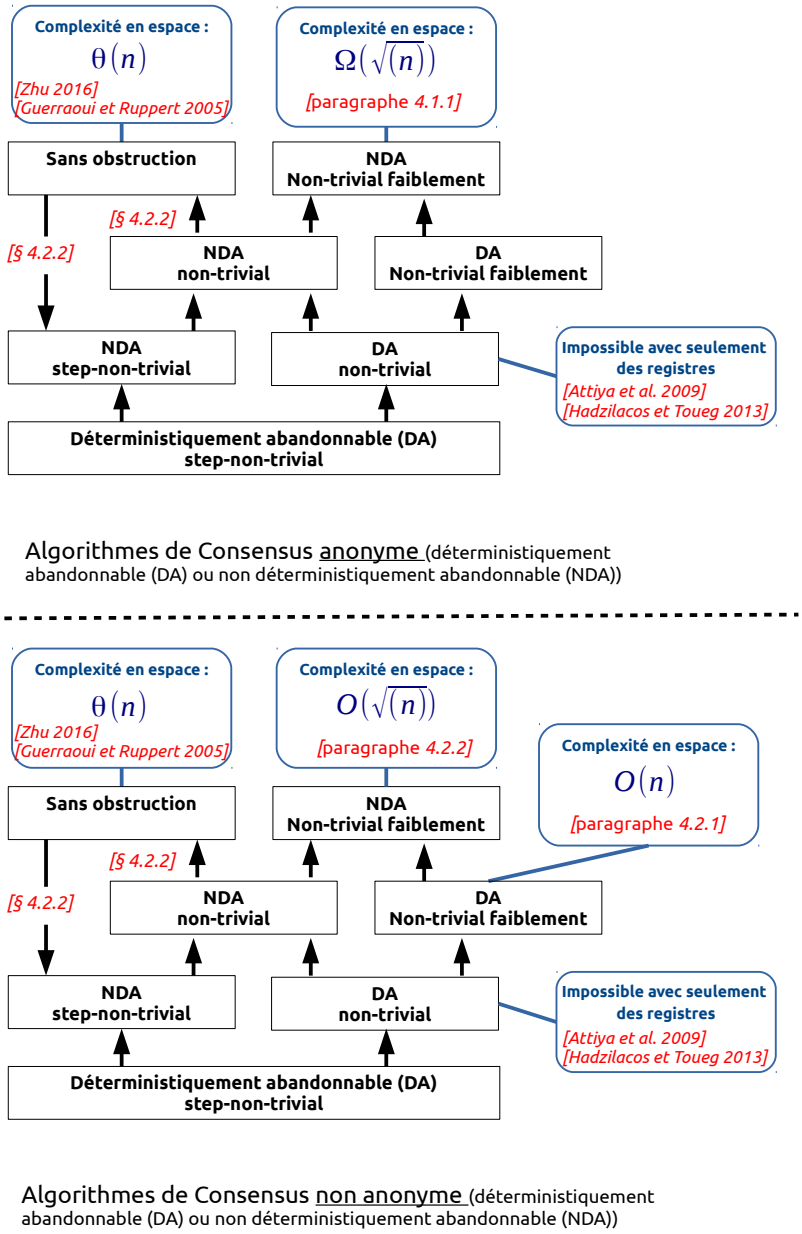

FIGURE 4.4 – Résultats sur la complexité spatiale des Consensus avec des propriétés d'abandons ou obstruction-free

4.2 Consensus avec la propriété d'abandon

Dans cette section, nous présentons les résultats concernant le consensus sans attente avec des propriétés d'abandon. La propriété d'abandon de l'objet peut être de plusieurs types, comme nous l'avons vu dans le Section 1.1. Deux types d'abandon ont été définis, l'abandon déterministe (Hadzilacos et Toueg [2013]) et l'abandon non-déterministe (Attiya *et al.* [2009] et Aguilera *et al.* [2007]). Pour rappel, dans le cas de l'abandon déterministe, le processus qui abandonne l'opération n'a pas changé l'état de l'objet, tandis que dans le cas d'abandon non-déterministe, le processus qui abandonne a peut-être changé l'état de l'objet.

A cette propriété d'abandon de l'objet s'ajoute une propriété de non-trivialité sur l'implémentation qui permet de limiter le retour "abandon" que peuvent faire les processus.

La réponse "abandon" doit être faite si l'opération abandonnée est en concurrence avec une autre opération. De plus, s'il s'agit d'une concurrence d'intervalle, la propriété suivante est ajoutée : si une opération ne se termine pas pour cause d'arrêt prématuré d'un processus, cette opération ne doit pas causer une infinité d'abandons.

La Figure 4.4 donne un résumé des relations entre les différentes propriétés étudiées ainsi que les résultats s'y rattachant. La flèche entre deux propriétés signifie que la propriété pointée par la flèche est conséquence de la propriété de départ de la flèche. Lorsque cette relation n'est pas triviale nous détaillons la transformation nécessaire pour passer d'une propriété à l'autre au paragraphe précisé sur la figure.

Dans la suite, l'ensemble des consensus étudiés ont la propriété de sans attente sauf dans le cas du consensus sans obstruction.

4.2.1 Abandon déterministe

Deux résultats sur l'implémentation du consensus avec abandon déterministe ont déjà été traités. Premièrement, il a été démontré par Attiya *et al.* [2009] que ce type d'objets ne peut pas être implémenté en utilisant que des primitives *Read* et *Write*. En effet, ils démontrent qu'il est possible d'implémenter un consensus sans attente pour 2 processus à l'aide du consensus avec abandon déterministe. Et, d'après la hiérarchie des objets proposée par Herlihy [1991], cette construction est impossible à l'aide de simples registres. La propriété de non-trivialité dans ce papier est basé sur une concurrence de pas de calcul, c'est-à-dire l'entrelacement de pas de calcul par plusieurs processus.

Le second résultat qui nous intéresse a été démontré par Hadzilacos et Toueg [2013]. Il confirme l'impossibilité démontré par Attiya *et al.* [2009] et la complète en prouvant qu'il est impossible d'implémenter un consensus avec abandon déterministe pour n processus avec un objet qui aurai un numéro de

consensus inférieur à n dans la hiérarchie de Herlihy [1991]. Ainsi, pour un n quelconque, il est nécessaire d'utiliser des primitives ayant un numéro de consensus infini, tel que le CAS.

L'implémentation d'un consensus avec un abandon déterministe et la non-trivialité ne permet donc pas de contourner l'impossibilité déjà présente avec le consensus sans attente. L'utilisation d'une primitive CAS permet trivialement l'implémentation d'un objet consensus (sans propriété d'abandon) donc à fortiori le consensus abandonnable. De plus la borne sur l'implémentation d'un consensus en limitant l'utilisation de la primitive CAS en cas de concurrence (traité dans la Section 4.1), ne dépend pas du caractère abandonnable ou non du consensus. En effet l'exécution est indistinguable d'une exécution solo pour les processus. L'ajout de la propriété d'abandon laisse donc les bornes inchangées.

Dans le cas de l'abandon déterministe, nous nous sommes donc intéressé à des objets plus faibles en affaiblissant la propriété de non-trivialité. La non-trivialité de l'abandon garantit que l'abandon se fait lors de la présence de concurrence et qu'une opération qui ne termine jamais à cause d'une panne d'un processus cause au plus un nombre fini d'abandon. L'affaiblissement se fait en supprimant la seconde partie. La non-trivialité faible garantit le fait que l'abandon d'une opération se fait uniquement lorsque l'opération est en concurrence avec une autre opération. Par conséquent, dans le cadre de la non-trivialité faible, la panne d'un processus peut causer l'abandon de toutes les autres opérations et ainsi rendre l'objet non tolérant aux pannes.

Nous montrons que dans le cas de l'abandon déterministe avec une non trivialité faible, l'implémentation du consensus en utilisant seulement des registres devient possible.

Consensus avec abandon déterministe faiblement non-trivial

L'algorithme 18 prouve la faisabilité d'un consensus déterministiquement abandonnable avec seulement des primitives $Read$ et $Write$ en considérant la propriété de non-trivialité faible. Cet algorithme a une complexité en espace de $O(n)$ et de $O(1)$ en pas de calcul.

Description Le registre D est le registre qui va contenir la valeur décidée. Le tableau R possède n registres, un pour chaque processus, de telle sorte qu'un processus n'écrit que dans un seul registre mais peut lire dans tous les registres. Dans chaque registre est stocké un bit, utilisé par les processus pour noter leur présence. 0 correspond à "je ne veux pas écrire dans D", et 1 à "je souhaite écrire dans D".

Un processus commence par vérifier si une valeur n'a pas déjà été décidée en lisant dans la variable D. Sinon, il annonce sa présence en mettant son registre attribué dans R à 1, indiquant aux autres processus sa volonté d'écrire

dans D. Puis, il lit l'ensemble des autres registres. Si dans l'un d'entre-eux il lit la valeur 1, cela veut dire qu'un autre processus veut écrire dans D. Comme ils ne peuvent pas se synchroniser, il remet à 0 son registre dans R, signalant son départ, et abandonne. Dans l'autre cas, il est le seul à souhaiter écrire dans D, il écrit sa valeur dans D et décide sa valeur.

Preuves Trivialement nous avons que :

Théorème 4.6 (Abandon déterministe)**.** *Une opération abandonnée ne change pas l'état de l'objet.*

et que :

Théorème 4.7 (Validité)**.** *Toutes les valeurs de sortie sont une des valeurs d'entrée (à l'exception de la réponse abandon).*

Et également que :

Théorème 4.8 (Sans attente)**.** *Toutes les opérations réalisées par un processus correcte retournent (une réponse qui peut être l'abandon).*

Théorème 4.9 (Accord)**.** *Deux processus ne retournent pas des valeurs différentes (sans considérer la réponse abandon).*

Démonstration. Par contradiction, supposons que deux valeurs différentes soient retournées, ce qui veut dire que deux valeurs différentes sont écrites dans le registre D. Soit p et q deux processus ayant écrit dans D des valeurs différentes. Si p a écrit dans D, alors au préalable il a lu dans $R[q]$ la valeur 0, donc q a écrit dans R après p. q lit $R[p]$ après avoir écrit dans $R[q]$, donc il lit après l'écriture de p dans $R[p]$. Or, p ne remet pas à zéro $R[p]$ parce qu'il n'exécute pas les lignes 6 et 7. Donc, q lit 1 dans $R[p]$ et n'écrit pas dans D à cause de la ligne 5. Il y a donc contradiction. □

Le lemme suivant est utilisé pour prouver la non-trivialité faible :

Lemme 4.10. *À une configuration C, si $R[p]$ est égale à 1, soit le processus p a une opération en cours d'exécution, soit p a écrit dans D.*

Démonstration. Soit C une configuration telle que $R[p]$ est égale à 1. p a déjà effectué la ligne 3. Si p n'est pas en cours d'exécution, alors il n'a pas retourné à la ligne 7, car sinon $R[p]$ serait égale à 0 à cause de la ligne 6. Il a donc retourné à la ligne 11, et donc précédemment exécuté la ligne 10. Donc si p n'est pas en cours d'exécution, p a écrit dans D. □

Théorème 4.11 (Non-trivialité faible)**.** *En absence de concurrence d'intervalle, aucune opération n'est abandonnée.*

Démonstration. Supposons un processus p qui retourne $\perp$ à la ligne 7 après avoir lu $R[q]$ égal à 1. Soit C la configuration juste avant qu'il ne retourne. Si à C, le registre D a une valeur différente de sa valeur initiale, alors un processus autre que p a écrit entre la lecture de D par p (de la ligne 2) et C. Donc, pendant l'exécution de p, il y a eu de la concurrence. Si à C, le registre D a toujours sa valeur initiale, alors, au moment de la lecture de $R[q]$, d'après le Lemme 4.10, le processus q avait une opération en cours. Donc pendant l'exécution de p, il y a eu de la concurrence. □

```
   Variables partagées :
   Registre D, initialement ⊥
   Tableau de registres R[1...n],initialement 0.
   Procédure : propose(v) par un processus p
1  val ← Read(D);
2  if val ≠ ⊥ then return val ;
3  Write(R[p], 1);
4  for i = 1 to n do
5  |  if i ≠ p ∧ Read(R[i]) ≠ 0 then
6  |  |  Write(R[p], 0);
7  |  |  return ⊥ ;
8  |  end
9  end
10 Write(D, v);
11 return v;
```

Algorithme 18: Algorithme non-trivial faible d'un consensus avec abandon déterministe, pour un processus p

4.2.2 Abandon non-déterministe

Pour contourner l'impossibilité de l'implémentation du consensus avec abandon déterministe avec seulement des primitives *Read* et *Write*, nous nous sommes également intéressés à des objets plus faibles en considérant la propriété de l'abandon non-déterministe. Nous montrons par la suite que l'implémentation d'un consensus avec abandon non-déterministe demande $\theta(n)$ objets de base qu'il garantisse la propriété de step-non-trivialité ou celle de non-trivialité. Ceci est démontré en prouvant l'équivalence de ces derniers avec un consensus sans obstruction. Puis, nous verrons qu'un affaiblissement de la propriété de non-trivialité permet l'implémentation d'un consensus avec abandon non-déterministe avec une complexité sous-linéaire en espace.

Dans cette sous-section, lorsque l'on parle d'objet abandonnable ou d'abandon, on considère sa variante non-déterministe.

Consensus non-trivial avec abandon non-déterministe

Pour prouver que l'implémentation des trois objets (consensus sans attente non-trivial avec abandon, consensus sans attente step-non-trivial avec abandon et consensus sans obstruction) sont équivalentes en terme de complexité, nous allons voir premièrement comment à partir d'un algorithme abandonnable non-trivialement construire un algorithme sans obstruction du consensus. Puis nous ferons la construction du consensus sans attente, abandonnable step-non-trivial vers un algorithme sans obstruction du consensus. Le passage du consensus sans attente step-non-trivial avec abandon vers celui sans attente non-trivial avec abandon est immédiat.

De l'implémentation du consensus abandonnable non-trivialement à une implémentation du consensus sans obstruction Le passage d'un consensus abandonnable à une implémentation sans obstruction de consensus non-abandonnable se fait de manière simple et intuitive. En effet, il suffit de prendre l'algorithme de l'objet abandonnable et de l'exécuter jusqu'à ce qu'il retourne une réponse différente de l'abandon, comme présenté dans l'Algorithme 19.

Les propriétés d'accord et d'intégrité du consensus proviennent directement des propriétés du consensus abandonnable. En ce qui concerne la propriété de progrès sans obstruction, c'est la propriété de la non-trivialité qui va l'assurer. Informellement, la propriété de non-trivialité assure que l'exécution d'une opération qui est abandonnée est en concurrence avec une autre et que si une opération reste inachevée à cause d'une panne, celle-ci ne va pas causer de manière infinie l'abandon d'autres opérations. Donc, si un processus s'exécute seul, il peut abandonner un nombre fini de fois, mais il retournera une réponse autre que l'abandon ($\perp$) s'il s'exécute suffisamment longtemps seul. Cela correspond à la propriété de sans obstruction. Nous avons donc que :

Théorème 4.12. *L'algorithme 19 implémente un consensus sans obstruction.*

Variables partagées :
abandonnable non-déterministe non-trivial consensus objet $ANTC$

Procédure : Propose(v_p)

```
1 res ← ⊥;
2 while res = ⊥ do
3 |   res ← ANTC(v_p);
4 end
5 return res;
```

Algorithme 19: Consensus sans obstruction

De l'implémentation du consensus sans obstruction à l'implémentation du consensus abandonnable step-non-trivial La construction

inverse est plus complexe car elle ne fait pas appel à l'algorithme sans obstruction d'un seul bloc mais le sectionne pour l'entremêler avec d'autres pas de calcul.

Plus précisément, considérons A_{obs} un algorithme sans obstruction d'un consensus qui utilise s variables partagées. Nous utilisons A_{obs} pour construire l'algorithme du consensus abandonnable step-non-trivial, appelé A_{ANTC}, qui utilise le même nombre de variables partagées. En plus des variables locales qui sont nécessaires pour A_{obs}, A_{ANTC} a besoin de $s+1$ variables locales, nommées $SV[1...s]$ et CP, initialisée à $\perp$. Les variables $SV[1...s]$ sont du même type que les s variables partagées utilisées par A_{obs} ; CP stocke un entier. Nous considérons que A_{obs} est composé de trois type de pas. Premièrement, les *pas de lecture* qui correspondent aux pas pour lesquels un processus lit une variable partagée. Deuxièmement, les *pas d'écriture* qui correspondent aux pas pour lesquels un processus écrit dans une variable partagée. Enfin, les *pas locaux* qui correspondent aux pas pour lesquels le processus n'accède qu'aux variables locales.

Une opération *propose* de A_{ANTC} procède ainsi. Le processus lit les variables partagées et copie les valeurs dans les variables locales $SV[1...s]$. Puis il lit la variable CP, qui contient la ligne du prochain pas à faire dans A_{obs}. S'il lit la valeur initiale, il exécute A_{obs} depuis le début, sinon il exécute l'algorithme A_{obs} depuis la ligne donnée par CP.

Chaque pas de A_{obs} est exécuté différemment suivant son type. À chaque *pas d'écriture*, le processus effectue les modifications sur les registres partagés et également sur les copies locales des registres. Pour les *pas de lecture*, le processus lit les registres partagés et compare la valeur stockée dans le registre partagé et la valeur du registre local correspondant. Si les valeurs sont les mêmes, le processus va au pas suivant de l'algorithme A_{obs}, sinon il écrit dans CP là où il s'arrête et retourne la réponse abandon. Les *pas locaux* sont faits de manière similaire à ceux de A_{obs}. A la fin, si le processus n'a pas abandonné, il retourne la valeur retournée par A_{obs}.

Si un processus invoque plusieurs instances d'une opération *propose*, entre chaque exécution des opérations *propose* par ce même processus toutes ses variables locales (à l'exception des variables $SV[1...s]$) doivent rester inchangées.

Comme précédemment, les propriétés d'accord et d'intégrité du consensus abandonnable step-non-trivial proviennent directement de celle du consensus sans obstruction. La propriété de sans obstruction assure que le processus qui s'exécute seul termine après un nombre fini de pas. De plus, de par la construction, si le processus ne s'exécute pas seul, il abandonnera. Donc, la propriété sans attente est assurée.

Comme un processus ne va abandonner que s'il lit une écriture faite par un autre processus après le début de son opération, l'exécution de l'opération abandonnée a bien été en concurrence de pas avec une autre opération. Nous avons donc que :

Théorème 4.13. *L'algorithme A_{ANTC} implémente un consensus abandonnable non-déterministe step-non-trivial.*

L'équivalence démontrée ci-dessous nous permet d'appliquer directement les résultats des bornes du consensus sans obstruction sur le consensus abandonnable de manière non déterministe. Ainsi d'après Zhu [2016] (pour la borne inférieure) et Guerraoui et Ruppert [2005] (pour la borne supérieure), que l'on soit dans un contexte anonyme ou non, la complexité en espace est de $\theta(n)$.

Consensus avec abandon non-déterministe faiblement non-trivial

Dans la section précédente, nous avons vu qu'il est nécessaire et suffisant d'implémenter un consensus pour n processus non-déterministiquement abandonnables à l'aide de n registres. Nous allons voir qu'il est possible en affaiblissant cette fois-ci la propriété de non-trivialité de passer sous cette borne en présentant un algorithme de consensus avec abandon non déterministe et la propriété de non-trivialité faible (*i.e.* l'abandon d'une opération se fait lors de la présence de concurrence avec une autre opération) dont la complexité spatiale est sous-linéaire.

Description de l'algorithme L'algorithme utilise un registre ID qui stocke un identifiant de processus, il va servir à désigner le processus qui a le droit d'écrire dans le tableau R. Le tableau de registres R sert à déterminer la valeur décidée qui sera écrite dans la dernière case du tableau. Chaque registre de R stocke un couple comportant un identifiant et une valeur.

Le processus vérifie en lisant dans la dernière case du tableau R, la présence d'une valeur décidée et retourne cette valeur si elle existe à la ligne 1. Puis, si ce n'est pas le cas, il vérifie la présence d'autres processus en lisant la valeur de ID à la ligne 3. Si un autre processus a déjà écrit son identifiant, il abandonne ; Sinon, il écrit son identifiant dans le registre ID et adopte une valeur qu'il va essayer de faire élire, en exécutant l'Algorithme 21. La valeur adoptée correspond à sa valeur si aucun processus n'a encore écrit dans R. Sinon, la valeur adoptée est la valeur du couple majoritairement présent dans R. En cas d'égalité entre plusieurs couples, c'est la valeur d'un des couples ex æquo, écrit dans le registre ayant l'index le plus grand, qui l'emporte.

Puis, tant que sa valeur est écrite dans ID (vérification de la ligne 8) et que la dernière case du tableau R a sa valeur initiale (vérification de la ligne 13), le processus écrit dans le registre du tableau R ayant le plus petit index et ne comportant pas le couple correspondant à la valeur adoptée et son identifiant. Si à un moment donné il découvre que son identifiant n'est plus écrit dans ID, il abandonne. Sinon, quand la dernière case du tableau R est écrite, il retourne la valeur décidée.

Variables partagées :
registre ID, initialement $\perp$
tableau de registres $R[1 \dots b]$ avec $n < \frac{b^2-3b}{2} + 3$. Initialement $< \perp, \perp >$

Procédure : $propose(v)$

```
1  < v_b, id_b > ← Read(R[b]) ;
2  if < v_b, id_b > ≠ < ⊥, ⊥ > then return v_b ;
3  if Read(ID) ≠ ⊥ then return ⊥ ;
4  Write(ID, id) ;
5  << v_1, id_1 > ... < v_b, id_b >> ← Read(R);
6  if (t ← AdoptValue(< v_1, id_1 > ... < v_b, id_b >)) ≠ ⊥ then v ← t ;
7  while R[b] == < ⊥, ⊥ > do
8      if Read(ID) ≠ id then return ⊥ ;
9      << v_1, id_1 > ... < v_b, id_b >> ← Read(R);
10     pos ← min(i | < v_i, id_i > ≠ < v, id >);
11     Write(R[pos], < v, id >) ;
12 end
13 return v ;
```

Algorithme 20: Non anonyme Consensus abandonnable faiblement non-trivial

Procédure : $AdoptValue(< v_1, id_1 > ... < v_b, id_b >)$

```
1  if v_1 == ⊥ then return ⊥;
2  c_max ← 0;
3  for i = 1 to b do
4      c ← 0;
5      if v_i ≠ ⊥ then
6          for j = 1 to b do
7              if < v_i, id_i > == < v_j, id_j > then c++;
8          end
9      end
10     if c_max ≤ c then < c_max, v_strong > ← < c, v_i >;
11 end
12 return v_strong;
```

Algorithme 21: Fonction AdoptValue

Preuves Trivialement nous avons que :

Théorème 4.14 (Validité). *Toutes les valeurs de sortie sont une des valeurs d'entrée.*

Lemme 4.15. *Si un processus p écrit le couple $< v, p >$ dans le registre $R[i]$, alors il existe une configuration C qui précède cette écriture telle que dans C, les valeurs des registres $R[1]$ à $R[i-1]$ sont égales à $< v, p >$ et la valeur stockée dans ID est p.*

Démonstration. Soit p un processus qui écrit le couple $< v, p >$ dans le registre $R[i]$. Pour cela, p a lut précédemment son identifiant dans le registre ID et le couple $< v, p >$ dans les registres $R[1]$ à $R[i-1]$. Soit C la configuration qui précède immédiatement la lecture de ID par p. Supposons par contradiction qu'à la configuration C, soit ID ne contient pas p, soit un registre $R[j]$ avec $j \in \{1, \dots i-1\}$ ne contient pas $< v, p >$. Cela veut dire qu'entre la configuration C et la lecture par p, un processus a écrit p dans ID ou $< v, p >$ dans $R[j]$. Cependant, seul p peut écrire son identifiant. Étant donné que p ne fait aucune écriture entre la configuration C et la lecture de ID ou de $R[j]$, il y a une contradiction. □

Pour toute exécution E, soit $C_{i,1}$ la configuration (si elle existe) qui précède immédiatement la première opération d'écriture dans le registre $R[i]$ avec $i \in \{1, \dots b\}$. Soit $p_{i,1}$ le processus qui effectue cette écriture et $v_{i,1}$ la valeur écrite. Similairement, soit $C_{i,2}$ la configuration (si elle existe) qui précède immédiatement la première opération d'écriture dans le registre $R[i]$ d'une valeur différente de $v_{i,1}$ avec $i \in \{1, \dots b\}$. Soit $p_{i,2}$ le processus qui effectue cette écriture et $v_{i,2}$ la valeur écrite. Enfin, pour $x \in \{1, 2\}$ et $i \in \{1, \dots b\}$, soit $C'_{i,x}$ la première configuration telle que les registres $R[1]$ à $R[i-1]$ contiennent le couple $< v_{i,x}, p_{i,x} >$ et la valeur du registre ID soit égal à $p_{i,x}$.

Lemme 4.16. *Soit $i \in [1, b]$. Dans toute exécution E où $C_{i+1,1}$ existe, s'il a une configuration $C'_{i,2}$, alors elle précède la configuration $C'_{i+1,1}$.*

Démonstration. Soit une exécution E dans laquelle $C_{i+1,1}$ existe. D'après le Lemme 4.15, la configuration $C'_{i+1,1}$ existe et par définition, à $C'_{i+1,1}$, le registre ID contient l'identifiant de $p_{i+1,1}$ et les registres $R[1]$ à $R[i]$ contiennent $< v_{i+1,1}, p_{i+1,1} >$. Supposons que $C'_{i,2}$ existe et suive la configuration $C'_{i+1,1}$. Nous allons prouver une contradiction sur le fait que $C'_{i,2}$ existe.

Soit S_b, l'ensemble des processus ayant écrit dans ID avant le processus $p_{i+1,1}$. Cet ensemble est bien défini car ID est un registre atomique et que chaque processus écrit au plus une fois dans celui-ci. Dans un premier temps, supposons que $p_{i,2}$ appartienne à S_b. Après $C'_{i+1,1}$, la valeur du registre ID est différente de n'importe quel identifiant des processus de S_b. Comme chaque processus écrit une seule fois son identifiant dans ID, il ne peut pas y avoir

une configuration qui suit $C'_{i+1,1}$ où ID contient l'identifiant d'un processus de S_b.

Soit S_a, l'ensemble des processus qui écrivent dans ID après $p_{i+1,1}$. Supposons maintenant que $p_{i,2} \in S_a$. Étant donné qu'à la configuration $C'_{i+1,1}$ le registre ID contient l'identifiant de $p_{i+1,1}$, ces processus écrivent dans ID après la configuration $C'_{i+1,1}$. Donc, ils écrivent dans les registres $R[1 \dots b]$ après $C'_{i+1,1}$. Par la suite, nous allons prouver que la valeur adoptée par les processus de S_a est $v_{i+1,1}$. Rappelons qu'un processus adopte une valeur une seule fois au cours de l'algorithme et avant d'écrire dans n'importe quel registre $R[1] \dots R[b]$ (à la ligne 6 de l'Algorithme 20). Donc, il écrit la même valeur dans tous les registres qu'il écrit. La preuve se fait par induction sur le j-ème processus de S_a qui écrit dans $R[1]$.

Cas de base. Le premier processus de S_a qui écrit dans $R[1]$, nommé p_1, a soit lu les valeurs écrites par des processus de S_b ou par $p_{i+1,1}$. Si les processus de S_b ont écrit dans un sous-ensemble des registres $R[1]$ à $R[i]$, soit p_1 lit une majorité de couple écrit par $p_{i+1,1}$, soit il lit que des couples distincts (car tous les processus de S_b écrivent au plus une fois après $C'_{i+1,1}$ et chacune de leur paire a un identifiant différent). Dans les deux cas, la valeur adoptée par p_1 est $v_{i+1,1}$. En effet, soit la valeur correspond à la valeur trouvée dans le couple majoritairement lu, soit c'est celle écrite dans $R[i]$. Aucun processus de S_b écrit une valeur différente de $v_{i+1,1}$ dans $R[i]$ car à $C'_{i+1,1}$ $R[i] = v_{i+1,1}$ et $p_{i,2} \notin S_b$.

Pas d'induction Supposons que pour les j premiers processus de S_a qui écrivent dans $R[1]$, la valeur adoptée est $v_{i+1,1}$. Nous prouvons que le $j+1$ème processus de S_a, nommé p_{j+1}, qui écrit dans $R[1]$, adopte $v_{i+1,1}$. p_{j+1} peut seulement lire $< v_{i+1,1}, p_{i+1,1} >$, un couple distinct pour chaque processus de S_b, ou encore des couples écrits par les $j$ème premiers processus de S_a à écrire dans $R[1]$. Seuls les couples écrits par des processus de S_b peuvent avoir une valeur différente de $< v_{i+1,1}, p_{i+1,1} >$. Pour les mêmes raisons que dans le cas de base, p_{j+1} adopte la valeur adopté par $p_{i+1,1}$.

Les processus de S_a n'écriront jamais une valeur différente de $p_{i+1,1}$ dans $R[i]$. Ainsi, $p_{i,2} \notin S_a$ et $C'_{i,2}$ n'existe pas. □

Observation 4.17. *Par définition de $C'_{i,1}$ et $C_{i,1}$, à $C'_{i,1}$ les registres $R[i] \dots R[b]$ contiennent leur valeurs initiales.*

Lemme 4.18. *Soit $i \in \{2, \dots, b\}$. Pour toute exécution E où $C'_{i,1}$ et $C'_{i,2}$ existent, au moins $i-2$ processus écrivent pour la dernière fois entre $C'_{i,1}$ et $C'_{i,2}$.*

Démonstration. Supposons sans perte de généralité que la configuration $C'_{i,1}$ précède la configuration $C'_{i,2}$.

Comme à la configuration $C'_{i,1}$, la valeur du registre ID est égale à $p_{i,1}$ et que à $C'_{i,2}$ la valeur de ID est égal à $p_{i,2}$, $p_{i,2}$ a écrit son identifiant dans ID après $C'_{i,1}$. Donc, $p_{i,2}$ adopte sa valeur après $C'_{i,1}$ (à la ligne 6 de l'Algorithme 20).

Comme à $C'_{i,1}$, les registres $R[1]$ à $R[i-1]$ contiennent le couple $< v_{i,1}, p_{i,1} >$ écrit par $p_{i,1}$, et de par l'Observation 4.17, des processus doivent écrire une valeur différente dans un sous-ensemble des registres $R[1] \dots R[i-1]$. Sinon, $p_{i,2}$ adopterai une valeur identique à $p_{i,1}$, ce qui constituerai une contradiction.

Ainsi, entre les configurations $C'_{i,1}$ et $C'_{i,2}$, il y a un ensemble d'opérations d'écriture exécuté par des processus qui ont écrit leur identifiant dans ID avant la configuration $C'_{i,1}$. Chacun de ces processus écrit au plus une fois après $C'_{i,1}$ (lignes 8 à 11 de l'Algorithme 20) et comme les couples contiennent les identifiants de chaque processus, tous les couples écrits sont différents. Donc, comme $p_{i,2}$ ne lisent pas une majorité de $< v_{i,1}, p_{i,1} >$ dans les i_1 premiers registres, au moins $i-2$ processus distincts exécutent une opération d'écriture entre les configurations $C'_{i,1}$ et $C'_{i,2}$. C'est leur dernière écriture. □

Théorème 4.19 (Accord)**.** *Deux processus ne retournent pas des valeurs différentes.*

Démonstration. Supposons par contradiction qu'il existe une exécution E dans laquelle deux processus retournent des valeurs différentes. Donc, deux couples avec des valeurs différentes ont été écrits dans le registre $R[b]$. Ce qui implique que $\forall i \in \{1, \dots, b\}$, la configuration $C_{i,2}$ existe. Donc, d'après le Lemme 4.15, $\forall\ i \in \{1, \dots, b\}$, les configurations $C'_{i,1}$ et $C'_{i,2}$ existent.

D'après le Lemme 4.18, pour tous $i \in \{2, \dots, b\}$, au moins $i-2$ processus écrivent pour leur dernière fois entre les configurations $C'_{i,1}$ et $C'_{i,2}$. Pour $i \in \{2, \dots, b-1\}$, $C'_{i,1}$ précède $C'_{i+1,1}$ et donc $C'_{i+1,2}$. De plus, $C'_{i,2}$ précède $C'_{i+1,2}$ et également $C'_{i+1,1}$, d'après le Lemme 4.16. Par conséquent, pour tous $i, j \in \{2, \dots, b\}$ tel que $i \neq j$, l'ensemble des processus qui écrivent pour la dernière fois entre $C'_{i,1}$ et $C'_{i,2}$ n'intersecte pas l'ensemble des processus qui écrivent pour la dernière fois entre $C'_{j,1}$ et $C'_{j,2}$.

Donc, il y a au moins $\sum_2^b (i-2) + 2 = \frac{b^2-3b}{2} + 3$ processus distincts. Ce qui est une contradiction. □

Théorème 4.20 (Sans attente)**.** *Toutes les opérations réalisées par un processus correct retournent (potentiellement la réponse abandon).*

Démonstration. Soit p un processus dans la boucle aux lignes 7 à 11. Soit ID n'est pas égal à p, dans ce cas, trivialement, p va retourner. Soit ID est égal à p et p retourne si $R[b]$ a été écrit. Nous définissons le progrès de p comme le premier index i tel que dans le tableau R, $R[i]$ ne contient pas le couple écrit par p. Le progrès de p peut seulement décroître à cause de l'écriture d'autres processus eux-mêmes dans cette boucle. Soit au moins un de ces processus a écrit son identifiant dans ID après p, dans ce cas, ID n'est plus égal à p et p va retourner. Dans l'autre cas, ces processus n'ont pas leur identifiant dans le registre ID, c'était leur dernière écriture, de plus ils n'entreront plus dans la boucle à cause de la ligne 3. Comme le nombre de processus est borné, le nombre de boucles est également borné. Les autres cas sont triviaux. □

Lemme 4.21. *A toute configuration C, si $R[b] = \perp$ et $ID \neq \perp$, alors le processus avec l'identifiant écrit dans ID est en cours d'exécution.*

Démonstration. Soit une configuration C, telle que à C, $R[b] = \perp$ et $ID = p$. Si l'identifiant de p est écrit dans ID, alors le processus p a déjà exécuté la ligne 4. Il n'a donc pas retourné à la ligne 3. De plus, à C, aucun processus n'a écrit après lui dans $R[b]$ ni écrit une autre valeur dans ID, donc le processus n'a pas pu retourner aux lignes 8 ou 13. Le processus p est donc toujours en cours d'exécution. □

Théorème 4.22 (Non-trivialité faible)**.** *En absence de concurrence d'intervalle, aucune opération n'est abandonnée.*

Démonstration. Soit un processus p qui abandonne à la ligne 8. p a lu l'identifiant d'un autre processus q dans le registre ID. q a écrit dans le registre ID après p, donc l'exécution de l'opération de p est bien en concurrence avec celle de q.

Soit un processus p qui abandonne à la ligne 3. p a lu $R[b] = \perp$ puis $ID \neq \perp$. Nous allons prouver qu'entre la lecture de $R[b]$ et celle de ID, il existe une configuration C telle qu'à C, $R[b] = \perp$ et $ID \neq \perp$.

Supposons qu'à la configuration C' juste après sa lecture de ID, $R[b]$ est toujours égal à $\perp$. Prenons la configuration C, étant la première configuration après la lecture dans $R[b]$ par p et tel que $ID \neq \perp$. Cette configuration existe puisque p lit $ID \neq \perp$, de plus comme C précède C', à C $R[b] = \perp$. Enfin, par définition de C, C est entre la lecture de $R[b]$ et celle de ID par p.

Supposons maintenant qu'à la configuration C' juste après sa lecture de ID, $R[b]$ n'est pas égal à $\perp$. Prenons la configuration C, étant la dernière configuration telle que $R[b] = \perp$. Cette configuration est bien après la lecture de $R[b]$ faite par p, car p lit $\perp$ dans $R[b]$, de plus, comme à C', $R[b] \neq \perp$ et que le pas d'avant C' est la lecture de ID par p, C précède la lecture de ID par p. Finalement, en observant l'algorithme, il est trivial de voir qu'un processus avant d'écrire dans $R[b]$ écrit dans ID. Donc, le processus qui écrit dans $R[b]$ après C a écrit dans ID avant C. Ainsi, à C, $R[b] = \perp$ et $ID = p$.

Comme entre la lecture de $R[b]$ et celle de ID par p, il existe une configuration C telle qu'à C, $R[b] = \perp$ et $ID \neq \perp$, d'après le Lemme 4.21, p est en concurrence avec un autre processus. □

4.3 Résumé

Ce chapitre a été consacré aux résultats sur le k-accord et le consensus. Dans une première partie, nous avons présenté les résultats sur le k-accord intervalle-solo-rapide anonyme. Nous avons démontré une borne inférieure de $\Omega(\min(\sqrt{n/k}, \log(m/k)/\log\log(m/k)))$ sur le nombre d'objets de base différents devant être écrits dans le pire cas d'une opération s'exécutant seule, avec

n le nombre de processus et m le nombre de valeurs d'entrée possibles. À noter que cette preuve est valable également pour un k-accord sans obstruction ou abandonnable. Nous avons prouvé que cette borne peut être atteinte dans le cas k-accord intervalle-solo-rapide anonyme, en donnant un algorithme utilisant l'objet k-value-splitter défini et implémenté dans le chapitre précédent.

Dans la seconde partie de ce chapitre, nous avons donné nos résultats sur le consensus abandonnable. Pour le cas du consensus déterministiquement abandonnable, nous avons montré qu'avec la propriété de non-trivialité faible, il est possible d'implémenter le consensus déterministiquement abandonnable en utilisant seulement des primitives Read et Write. Pour le cas du consensus non déterministiquement abandonnable avec la propriété de non-trivialité ou de step-non-trivialité, nous avons prouvé leur équivalence en terme de complexités spatiale avec le consensus sans obstruction. Nous avons également proposé un algorithme sous-linéaire dans le cas du consensus non déterministiquement abandonnable en affaiblissant la propriété de non-trivialité à la non-trivialité faible.

Conclusion

La programmation d'objets concurrents sans attente, dans un système asynchrone, n'est pas aisé. Une limitation immédiate est l'impossibilité d'implémenter le consensus sans-attente avec seulement des registres.

Pour réaliser une implémentation sans attente d'objets concurrents, nous avons étudié deux types d'implémentations. Le premier type d'implémentations autorise l'abandon par les processus de leur opération en cours (en cas d'abandon une réponse spécifique est retournée). Le deuxième type d'implémentations limite l'utilisation des primitives puissantes mais coûteuses en terme de temps d'exécution (à savoir toutes les primitives à l'exception de $Read$ et $Write$).

Dans nos travaux nous sommes intéressés à l'implémentation d'objets ayant l'une ou l'autre, voir les deux propriétés : objets abandonnables ou/et objets solo-rapides. L'étude de la complexité en espace pour ces implémentations a également été au cœur de nos travaux.

Nous avons ainsi proposé l'implémentation d'objets concurrents, intervalle-solo-rapide non-trivialement, sans attente et déterministiquement abandonnables de manière non trivial à l'aide d'une construction universelle. Nous avons défini le k-value-splitter (brique de base pour l'implémentation du consensus et du k-accord). À l'aide du k-value-splitter, nous avons proposé une implémentation intervalle-solo-rapide optimale d'un k-accord. Nous avons également proposé plusieurs implémentations de consensus abandonnables.

Nous avons établi qu'un consensus intervalle-solo-rapide ou qu'un consensus non déterministiquement abandonnable et faiblement non trivial nécessite moins d'espace ($O(\sqrt{(n)})$) que le consensus sans obstruction ($O(n)$).
Un consensus non déterministiquement abandonnable et faiblement non trivial est certes sans attente mais le caractère de faible non trivialité le rend non tolérant aux pannes des processus.

Ainsi, le gain en terme d'espace est réduit. Finalement, l'avantage des solutions proposées est de permettre un retour au programmeur contrairement aux objets sans obstruction. Et d'autre part, de limiter l'utilisation de primitives tel que le CAS, par rapport à des solutions sans attente "traditionnelle", ayant des complexités moindres. Le bénéfice final, en terme de temps de calcul sur une machine, est alors très dépendant de la technologie existante.

Questions ouvertes Nous avons établi que l'implémentation d'objets perturbables intervalle-solo-rapide et déterministiquement abandonnables a une complexité en espace de $O(n)$. Pour l'implémentation d'objets perturbables step-solo-rapide, la borne sur l'espace prouvée par Attiya *et al.* [2009] est également $O(n)$. Il serait intéressant d'établir le coût d'une construction universelle intervalle-solo-rapide (non abandonnable). Par exemple, en transformant la construction universelle présentée dans ce manuscrit. Une autre piste serait de déterminer si l'implémentation d'un objet intervalle-solo-rapide déterministiquement abandonnable est plus économe en pas de calcul que l'implémentation uniquement intervalle-solo-rapide.

Il serait également intéressant de calculer le coût de la généricité de notre construction universelle. Quel serait le gain entre une structure de données implémentée spécifiquement de manière concurrente par rapport à son implémentation par notre construction universelle ? De plus, il serait intéressant de tester en environnement réel, notre construction universelle, et de comparer ses performances avec d'autres constructions que ne limitent pas l'utilisation des primitives CAS (par exemple celle proposée par Chuong *et al.* [2010]).

La comparaison de notre construction universelle avec des implémentations de mémoire transactionnelle serait également enrichissante. En effet, les objets abandonnables se comportent de manière similaire aux mémoires transactionnelles. Cependant, une mémoire transactionnelle peut encapsuler n'importe quel code séquentiel, elle est donc plus générique qu'une construction universelle. Là encore, une mesure du coût de la généricité de l'implémentation pourrait être intéressante.

Sur le consensus abandonnable nous avons d'une part une borne inférieure dans le cas anonyme ($\Omega(\min(\sqrt{n}, \log(m)/\log\log(m))))$ et nous avons des bornes supérieures dans le cas non anonyme ($O(n)$ et $O(\sqrt{(n)}))$. Il serait intéressant de fermer ces problèmes, par exemple en répondant à la question suivante : quelles sont les conditions pour réaliser une implémentation du consensus abandonnable ayant une complexité spatiale sous linéaire ? Nous avons établi qu'une telle implémentation existe : une implémentation non anonyme du consensus non déterministiquement abandonnable et faiblement non trivial. Il s'agit de l'implémentation du consensus abandonnable la plus faible que nous étudions dans ce manuscrit.

Enfin nous avons prouvé une équivalence de la complexité spatiale de 3 implémentations du consensus : consensus sans obstruction, consensus non déterministiquement abandonnable et non trivial, et consensus non déterministiquement abandonnable et step-non-trivial. D'autre part nous avons prouvé une borne inférieure valable pour les objets solo-rapide, abandonnable ou sans obstruction. Ainsi, les résultats sur les objets abandonnables sont similaires à ceux obtenus sur les objets sans obstruction. C'est pourquoi, nous espérons que certains mécanismes utilisés pour implémenter le k-value-splitter ou le conflict-detector puissent être utilisés pour améliorer la borne supérieure sur

le k-accord sans obstruction.

Bibliographie

Aguilera, Marcos K., Frolund, Svend, Hadzilacos, Vassos, Horn, Stephanie L. et Toueg, Sam, 2007. Abortable and query-abortable objects and their efficient implementation. Dans *the 26th ACM Symposium on Principles of Distributed Computing (PODC'07)*, pages 23–32. ISBN 978-1-59593-616-5. doi :10.1145/1281100.1281107.

Aspnes, James et Ellen, Faith, 2014. Tight bounds for adopt-commit objects. *Theory of Computing Systems*, 55(3) :451–474. doi :10.1007/s00224-013-9448-1.
URL http://dx.doi.org/10.1007/s00224-013-9448-1

Aspnes, James et Herlihy, M., 1990. Fast randomized consensus using shared memory. *J. Algorithms*, 11(3) :441–461. doi :10.1016/0196-6774(90)90021-6.
URL http://dx.doi.org/10.1016/0196-6774(90)90021-6

Attiya, Hagit, Guerraoui, Rachid, Hendler, Danny et Kuznetsov, Petr, 2009. The complexity of obstruction-free implementations. *J. ACM*, 56(4) :24 :1–24 :33. doi :10.1145/1538902.1538908.

Borowsky, Elizabeth et Gafni, Eli, 1993. Generalized FLP impossibility result for t-resilient asynchronous computations. Dans *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 91–100. doi :10.1145/167088.167119.
URL http://doi.acm.org/10.1145/167088.167119

Bouzid, Zohir, Raynal, Michel et Sutra, Pierre, 2015. Anonymous obstruction-free (n, k)-set agreement with n-k+1 atomic read/write registers. *CoRR*, abs/1507.00474.
URL http://arxiv.org/abs/1507.00474

Buhrman, Harry, Garay, Juan A., Hoepman, Jaap-Henki et Moir, Mark, 1995. Long-lived renaming made fast. Dans *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC

'95, pages 194–203. ACM, New York, NY, USA. ISBN 0-89791-710-3. doi : 10.1145/224964.224986.
URL http://doi.acm.org/10.1145/224964.224986

Capdevielle, Claire, Johnen, Colette, Kuznetsov, Petr et Milani, Alessia, 2015. On the Uncontended Complexity of Anonymous Consensus. Dans *OPODIS 2015*. Rennes, France.

Capdevielle, Claire, Johnen, Colette et Milani, Alessia, 2014. Solo-fast universal constructions for deterministic abortable objects. Dans *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, pages 288–302. doi : 10.1007/978-3-662-45174-8_20.
URL http://dx.doi.org/10.1007/978-3-662-45174-8_20

Capdevielle, Claire, Johnen, Colette et Milani, Alessia, 2016. On the Space Complexity of Conflict Detector Objects. Dans *Brief Announcements, DISC 2016*. Pari, France.

Chaudhuri, Soma, 1993. More choices allow more faults : Set consensus problems in totally asynchronous systems. *Inf. Comput.*, 105(1) :132–158. doi :10.1006/inco.1993.1043.
URL http://dx.doi.org/10.1006/inco.1993.1043

Chuong, Phong, Ellen, Faith et Ramachandran, Vijaya, 2010. A universal construction for wait-free transaction friendly data structures. Dans *the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*, pages 335–344. ISBN 978-1-4503-0079-7. doi :http://doi.acm.org/10.1145/1810479.1810538.
URL http://doi.acm.org/10.1145/1810479.1810538

Crain, Tyler, Imbs, Damien et Raynal, Michel, 2013. Towards a universal construction for transaction-based multiprocess programs. *Theor. Comput. Sci.*, 496 :154–169. doi :10.1016/j.tcs.2012.09.011.
URL http://dx.doi.org/10.1016/j.tcs.2012.09.011

Delporte-Gallet, Carole, Fauconnier, Hugues, Kuznetsov, Petr et Ruppert, Eric, 2015. On the space complexity of set agreement. *CoRR*, abs/1505.02690.
URL http://arxiv.org/abs/1505.02690

Fich, Faith, Herlihy, Maurice et Shavit, Nir, 1998. On the space complexity of randomized synchronization. *J. ACM*, 45(5) :843–862. doi :10.1145/290179.290183.

FISCHER, Michael J., LYNCH, Nancy A. et PATERSON, Michael S., 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2) :374–382.

GAFNI, Eli, 1998. Round-by-round fault detectors (extended abstract) : Unifying synchrony and asynchrony. Dans *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 143–152. ACM, New York, NY, USA. ISBN 0-89791-977-7. doi : 10.1145/277697.277724.
URL http://doi.acm.org/10.1145/277697.277724

GAFNI, Eli et GUERRAOUI, Rachid, 2011. Generalized universality. Dans *CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*, pages 17–27. doi :10.1007/978-3-642-23217-6_2.
URL http://dx.doi.org/10.1007/978-3-642-23217-6_2

GELASHVILI, Rati, 2015. On the Optimal Space Complexity of Consensus for Anonymous Processes. Dans Yoram Moses et Matthieu Roy, rédacteurs, *DISC 2015*, tome LNCS 9363 de *29th International Symposium on Distributed Computing*. Toshimitsu Masuzawa and Koichi Wada, Springer-Verlag Berlin Heidelberg, Tokyo, Japan. doi :10.1007/978-3-662-48653-5_30.
URL https://hal.archives-ouvertes.fr/hal-01207153

GIAKKOUPIS, George, HELMI, Maryam, HIGHAM, Lisa et WOELFEL, Philipp, 2013. An $O(\sqrt{n})$ space bound for obstruction-free leader election. Dans *Proceedings of the 27th International Symposium on Distributed Computing (DISC)*, pages 46–60.

GUERRAOUI, Rachid et KAPAŁKA, Michal, 2008. On obstruction-free transactions. Dans *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 304–313. ACM. ISBN 978-1-59593-973-9. doi :http://doi.acm.org/10.1145/1378533.1378587.
URL http://doi.acm.org/10.1145/1378533.1378587

GUERRAOUI, Rachid et RUPPERT, Eric, 2005. What can be implemented anonymously ? Dans *Proceedings of the 19th International Conference on Distributed Computing*, DISC'05, pages 244–259. Springer-Verlag, Berlin, Heidelberg. ISBN 3-540-29163-6, 978-3-540-29163-3. doi :10.1007/11561927_19.
URL http://dx.doi.org/10.1007/11561927_19

GUERRAOUI, Rachid et RUPPERT, Eric, 2007. Anonymous and fault-tolerant shared-memory computing. *Distributed Computing*, 20(3) :165–177. doi : 10.1007/s00446-007-0042-0.
URL http://dx.doi.org/10.1007/s00446-007-0042-0

HADZILACOS, Vassos et TOUEG, Sam, 2013. On deterministic abortable objects. Dans *the 2013 ACM Symposium on Principles of Distributed Computing (PODC'13)*, pages 4–12. ISBN 978-1-4503-2065-8. doi :10.1145/2484239.2484241.

HERLIHY, Maurice, 1991. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1) :124–149.

HERLIHY, Maurice, LUCHANGCO, Victor et MOIR, Mark, 2003. Obstruction-free synchronization : Double-ended queues as an example. Dans *the 23rd International Conference on Distributed Computing Systems (ICDCS'03)*, pages 522–529.

HERLIHY, Maurice et MOSS, J. Eliot B., 1993. Transactional memory : Architectural support for lock-free data structures. Dans *Proceedings of the 20th Annual International Symposium on Computer Architecture. San Diego, CA, May 1993*, pages 289–300. doi :10.1145/165123.165164.
URL http://doi.acm.org/10.1145/165123.165164

HERLIHY, Maurice et SHAVIT, Nir, 1999. The topological structure of asynchronous computability. *J. ACM*, 46(6) :858–923. doi :10.1145/331524.331529.
URL http://doi.acm.org/10.1145/331524.331529

HERLIHY, Maurice P. et WING, Jeannette M., 1990. Linearizability : A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3) :463–492. doi :10.1145/78969.78972.
URL http://doi.acm.org/10.1145/78969.78972

JAYANTI, Prasad, TAN, King et TOUEG, Sam, 2000. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.*, 30(2) :438–456.

LAMPORT, Leslie, 1974. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8) :453–455. doi :10.1145/361082.361093.
URL http://doi.acm.org/10.1145/361082.361093

LAMPORT, Leslie, 1987. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1) :1–11. doi :10.1145/7351.7352.
URL http://doi.acm.org/10.1145/7351.7352

LOUI, M.C. et ABU-AMARA, H.H., 1987. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4 :163–183.

LUCHANGCO, Victor, MOIR, Mark et SHAVIT, Nir, 2003. On the uncontended complexity of consensus. Dans *the 17th International Symposium on Distributed Computing (DISC'03)*, pages 45–59.

MOIR, Mark et ANDERSON, James H., 1995. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Program.*, 25(1) :1–39. doi :10.1016/0167-6423(95)00009-H.
URL http://dx.doi.org/10.1016/0167-6423(95)00009-H

PERELMAN, Dmitri, FAN, Rui et KEIDAR, Idit, 2010. On maintaining multiple versions in STM. Dans *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 16–25. doi :10.1145/1835698.1835704.
URL http://doi.acm.org/10.1145/1835698.1835704

SAKS, Michael E. et ZAHAROGLOU, Fotios, 2000. Wait-free k-set agreement is impossible : The topology of public knowledge. *SIAM J. Comput.*, 29(5) :1449–1483. doi :10.1137/S0097539796307698.
URL http://dx.doi.org/10.1137/S0097539796307698

SUTRA, Pierre, RIVIÈRE, Étienne et FELBER, Pascal, 2014. *Principles of Distributed Systems : 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings*, chapitre A Practical Distributed Universal Construction with Unknown Participants, pages 485–500. Springer International Publishing, Cham. ISBN 978-3-319-14472-6. doi :10.1007/978-3-319-14472-6_32.
URL http://dx.doi.org/10.1007/978-3-319-14472-6_32

YANG, Jiong, NEIGER, Gil et GAFNI, Eli, 1998. Structured derivations of consensus algorithms for failure detectors. Dans *In Proceedings of the 17th ACM Symposium on Principles of Distributed Computing*, pages 297–306. ACM Press.

ZHU, Leqi, 2016. A tight space bound for consensus. http://www.cs.toronto.edu/~lezhu/.