Structures de données pour environnements distribués à grande échelle

Grégoire Bonin

UNIVERSITE BRETAGNE LOIRE MATHSTIC

UNIVERSITÉ DE NANTES

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE NANTES
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Grégoire BONIN

« Structures des données pour environnements distribués à grande échelle »

Thèse présentée et soutenue à Nantes, le 17 Novembre 2021
Unité de recherche : LS2N, UMR 6004
Thèse N° :

Rapporteurs avant soutenance :

Bernadette Charron-Bost	Directrice de recherche CNRS, LIX, Ecole Normale Supérieure
Stéphane Devismes	Professeur, Université de Picardie Jules Verne

Composition du Jury :

Président :	François Taiani	Professeur, Université de Rennes 1
Examinateurs :	François Taiani	Professeur, Université de Rennes 1
	Corentin Travers	Maître de conférence, Labri/ENSEIRB Bordeaux
Directeur de thèse :	Achour Mostéfaoui	Professeur, Université de Nantes
Encadrant de thèse :	Matthieu Perrin	Maître de conférence, Université de Nantes

Remerciements

Je tiens à remercier tout d'abord les membres du jury et plus particulièrement les rapporteurs pour leurs remarques qui m'ont été fort utiles dans la finition de ce manuscrit.

Un grand merci à mes encadrants, Achour et Matthieu, pour tout leur soutien durant la thèse, surtout pendant les périodes de doute qui ont pu parsemer ces quatre années : Je ne serais peut-être pas arrivé au bout sans votre aide.

Je tiens aussi à remercier tous les membres de l'équipe GDD pour leur gentillesse et leur accueil, ainsi que les personnels administratifs du laboratoire - les assistantes d'équipe et de laboratoire, et le service informatique - sans qui cette thèse aurait été bien plus compliquée.

Je tenais enfin à remercier ma famille et mes amis pour leur soutien, ainsi que les membres de mes différents groupes de musique, qui m'ont permis de décompresser et de m'évader tout au long de ma thèse.

Sommaire

Chapitre 1

Introduction

Le projet O'Browser vise à proposer un nouveau modèle de programmation lié à un nouveau modèle de calcul distribué : les Foglets. Le but est de proposer un environnement de programmation via navigateurs internet interposés : la base de ce modèle est une forte décentralisation, ainsi qu'un ensemble de participants dynamique potentiellement infiniment croissant.

Afin d'implémenter des objets de base (registre, pile, file, liste par exemple) qui permettront la création d'applications distribuées dans ce modèle, il faut être capable de maintenir la cohérence des données. Dans ce but, la réplication des données offre une robustesse aux objets : si un objet était géré par un unique participant en local, le départ ou la perte de celui-ci entraînerait la perte des données de cet objet chez tous les participants. Chaque participant maintient donc un réplicat de l'objet localement, mais cela peut amener des problèmes d'incohérence des données si les modifications locales sur l'objet ne sont pas propagées chez tous les autres réplicats.

Dans ce système, un objet partagé devrait se comporter comme un unique objet sur lequel les processus effectuent des opérations atomiques (opérations qui apparaissent comme s'étant effectuées en un unique point et donc ininterruptible) : ceci est représenté par les *critères de cohérence* forts comme la linéarisabilité [HW90] ou la cohérence séquentielle [AW94]. Cette cohérence forte nécessite de mettre en place des mécanismes de synchronisation coûteux afin de faire en sorte que les opérations apparaissent comme s'étant exécutées de manière séquentielle.

Dans ces cas, et si l'on prend l'exemple du registre partagé dans un système à passage de messages (registre de mémoire répliqué chez tous les participants et dont la mise à jour cohérente se fait via l'envoi de messages entre les participants), il existe des situations dans lesquelles le temps de réponse des opérations sur le registre (lecture ou écriture) est proportionnel à la latence du réseau [LS88], ce qui, à l'échelle du réseau entier, et dans le cas d'une forte concurrence sur les

opérations, est très coûteux en terme de complexité temporelle. Ce coût étant souvent inacceptable compte tenu de la perte de temps engendrée, des critères de cohérence faibles ont donc été introduits pour palier ces problèmes : la cohérence à terme [Vog09] ou la cohérence d'écriture [PMJ15], par exemple, permettent de proposer une cohérence des objets au prix d'une période « d'incertitude » due à la concurrence des opérations, mais résolue à terme.

Un des points primordiaux à prendre en compte dans la recherche de la cohérence est la vivacité. En effet si l'obtention de la cohérence forte signifie que certains participants se retrouvent bloqués ou dans une situation de famine, cela est problématique : il faut donc également respecter certaines *conditions de progression*. La condition de progression la plus forte est la *wait-freedom* [Her91], qui spécifie qu'aucun participant ne peut être bloqué indéfiniment.

L'un des buts du projet O'browser est de pouvoir proposer aux développeurs un moyen d'implémenter l'ensemble des objets permettant de programmer une application sans avoir à se soucier des problèmes de communication et de synchronisation sous-jacents. Le but est donc de proposer des constructions universelles [Ray17] qui permettraient l'implémentation de tout objet dans les foglets. Une construction universelle est un algorithme qui, pour tout objet doté d'une spécification séquentielle, implémente ledit objet dans le système. Un objet est dit universel dans un système, si l'on peut implémenter une construction universelle dans ce système à l'aide d'un nombre quelconque d'instances de cet objet.

A l'instar de certains modèles de calcul décentralisés déjà existants comme le pair-à-pair [Mil+02], les foglets ont une spécificité (comparable aux systèmes de *multi-threading*) : les différents participants (similaires à un thread) peuvent rejoindre l'application à tout moment, ou quitter l'application à tout moment. Cela est utile pour modéliser les pannes franches : un participant qui tomberait en panne (et donc ne participerait plus) peut être considéré comme ayant quitté le système. Les modèles de calcul classiques étant souvent limités à un certain nombre n de processus, il nous faut explorer les propriétés des modèles ouverts [GMT01]. Parmi ces modèles, il existe plusieurs variations portant sur le nombre de processus pouvant arriver à un certain point (arrivées finies, bornées, infinies), et de cela vont dépendre les propriétés de calculabilité et de complexité de ces modèles et de différents objets dans ces modèles. En effet un objet universel dans le modèle classique ne l'est plus forcément si de nouveaux processus peuvent

arriver pendant l'exécution.

Le modèle de calcul correspondant aux Foglets est donc celui d'un système ouvert pouvant être exprimé par le modèle suivant : le modèle wait-free par passage de message avec pannes franches, à arrivées infinies [1].

La question principale à laquelle nous répondons dans cette thèse est donc la suivante :

Problématique : Comment régler les problèmes spécifiques aux systèmes ouverts dans l'implémentation des structures de données partagées ?

Le problème étant complexe, et pour une meilleure compréhension, nous diviserons la réponse en deux parties, en ne faisant varier qu'un seul paramètre à chaque fois, afin de mieux en exposer les subtilités

Calculabilité : Du point de vue de la calculabilité, il est intéressant d'étudier les changements apportés par les arrivées finies et infinies : en effet certains algorithmes peuvent devenir non valides ou ne plus terminer si le nombre n de participants pour lesquels ils ont été conçus est dépassé pendant une exécution, et donc tous les objets possédant un numéro de consensus infini dans la hiérarchie wait-free de Herlihy ne sont plus forcément équivalents dans les différents modèles d'arrivées.

Complexité : Certains sous-modèles proposant des restrictions sur les algorithmes et implémentations d'objets distribués permettent un fonctionnement et une utilisation simplifiée, qui serait adaptée au modèle de O'browser, mais à quel coût ? Du point de vue de la complexité en mémoire locale et de la complexité en nombre de messages, il est donc intéressant d'étudier quels sont les avantages ou inconvénients desdits sous-modèles par rapport au modèle général proposé, sous certaines conditions de synchronisation et de cohérence des données.

Thèse : L'utilisation de constructions universelles en système ouvert permet de proposer une implémentation simple pour tout objet en cachant les mécanismes

1. Ce système ne prend pas en compte les adversaires byzantins

de maintien de la cohérence et de coordination.

Contributions : Un problème central des systèmes distribués étant le consensus, la première contribution de cette thèse est de répondre à la question de l'universalité du consensus dans le cas du modèle le moins restrictif possible : le modèle d'arrivées infinies. La seconde contribution de cette thèse est l'extension de la hiérarchie wait-free de Herlihy [Her91] aux modèles ouverts en comparant la puissance de synchronisation de divers objets dans chaque étage de cette hiérarchie. La troisième contribution de cette thèse est l'étude des constructions universelles dans lesdits modèles, et l'introduction d'une construction universelle $UC[k]$ permettant de troquer un gain en complexité spatiale contre une perte en terme de complexité de communication, et vis-versa, en jouant uniquement sur le paramètre k.

Cette thèse a donné lieu à 4 publications en conférences (ainsi que deux publications en journal actuellement en attente)

- Août 2019, International Conference on Parallel Computing Technologies (PaCT) 2019, [BMP19a](best paper) : comparaison des modèles opérationnel et wait-free en complexité.
- Octobre 2019, International Symposium on Distributed Computing (DISC) 2019, [BMP19b] : preuve de l'universalité du consensus dans le cas d'arrivées infinies.
- Juillet 2020, Symposium on Principles of Distributed Computing (PODC) 2020 [PMB20] : extension de la hiérarchie de Herlihy aux modèles ouverts.
- 2021, Journal of Parallel and Distributed Computing : en attente, extension de [BMP19a].
- 2021, Distributed Computing Journal : en attente, extension de [PMB20].
- Septembre 2021, 23èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications ALGOTEL 2021, [PMB21] : diffusion de l'article [PMB20] en français.

Plan : La première partie de cette thèse est composée des chapitres 2 et 3 dans lesquels sont introduits respectivement des définitions préliminaires et un état de l'art permettant de situer les contributions plus précisément, puis une définition

plus précise des modèles, ainsi que les différences entre les modèles utilisés dans les partie suivantes.
La deuxième partie est composée des chapitres 4 et 5, qui présentent les résultats sur l'étude de l'universalité dans les modèles ouverts en communication par mémoire partagée, avec d'abord le cas du consensus dans le modèle d'arrivées infinies (chapitre 4), puis ensuite l'extension de la hiérarchie de Herlihy aux différents modèles ouverts (chapitre 5).
La troisième partie de la thèse est composée du chapitre 6 et s'intéresse cette fois à la complexité dans les systèmes à communication par passage de message.

Chapitre 2

Etat de l'art

2.1 Définitions préliminaires

A la base de tout système de calcul distribué se trouvent les différents participants ici appelés processus. Ces processus ont la capacité d'exécuter des opérations, de communiquer avec les autres processus, d'accéder et modifier la mémoire (locale ou partagée selon les modèles).

Définition 1 (Processus)**.** *Blocs de base du système distribué, les processus sont généralement nommés* $p_0, p_1, p_2, etc \in \Pi$ *selon le nombre total de processus présents dans le système.*

Dans les systèmes distribués, plusieurs types d'objets partagés peuvent être proposés pour fournir aux processus des abstractions de niveau supérieur. Il existe deux principaux types d'objets. D'une part, les objets à usage unique (ou tâches), comme le consensus et le renommage, sont une généralisation des fonctions dans les systèmes séquentiels, où chaque processus propose une entrée et décide une sortie. Les tâches sont spécifiées par une relation binaire qui relie les vecteurs d'entrée aux vecteurs de sortie admis. D'autre part, les objets persistants, tels que les registres et les files, sont une généralisation des structures de données en programmation séquentielle, visant à stocker et organiser les données en mémoire.

Définition 2 (Objet partagé)**.** *Un objet partagé O est défini par un état initial, un ensemble d'opérations, et une spécification qui relie un état d'entrée de l'objet et une opération de l'objet à un état de sortie.*

Parmi ces opérations, il existe certaines catégories selon l'effet ou le retour de l'opération. Deux grandes catégories sont celles des opérations d'écriture (ou de mise à jour) et des opérations de lecture (ou de requête). Une opération d'écriture modifie l'état de l'objet sans retourner de valeur (incrémenter un compteur sans

retourner la valeur par exemple), tandis qu'une opération de lecture retourne une valeur sans modifier l'état de l'objet (retourner la valeur contenue dans le compteur). Il existe certaines opérations n'appartenant pas à ces deux catégories, en modifiant à la fois l'état de l'objet, tout en retournant aussi une valeur (incrémenter le compteur, puis retourner la valeur du compteur). Dans cette thèse, si une opération modifie l'état de l'objet (même si une valeur est retournée), on appellera cette opération une opération d'écriture ou de mise à jour par abus de langage.

Nous ne nous occuperons ici que d'objets déterministes.

Afin de décrire le déroulement d'une application distribuée ou le comportement d'un objet au cours du temps, il nous faut définir la notion d'exécution distribuée.

Définition 3 (Exécution distribuée)**.** *Une exécution α est une séquence (finie ou infinie) d'étapes (ou pas de calcul) d'un algorithme, chacune prise par un processus de Π. Une étape d'un processus correspond à l'exécution d'une instruction matérielle. On dit qu'un processus p_i* arrive *dans une exécution lors de sa première étape dans cette exécution. De plus, il est possible qu'un processus arrête d'effectuer des pas de calcul à un certain moment de l'exécution (auquel cas nous disons que ce processus est* tombé en panne*), ou même qu'un processus n'effectue aucune étape pendant toute une exécution ($|\Pi|$ n'est qu'une limite supérieure du nombre de processus participants). Une exécution vide est notée ε. Une exécution β est une extension de α si α est un préfixe de β.*

Remarquez que, bien que le nombre de processus dans une exécution puisse être potentiellement infini, le nombre de processus qui sont arrivés dans le système à n'importe quelle étape est fini.

Définition 4 (Configuration)**.** *Une configuration C est composée de l'état local de chaque processus dans Π et de l'état interne de chaque objet partagé , y compris les registres de lecture/écriture. Pour une exécution finie α, on note $C(\alpha)$ la configuration obtenue à la fin de α.*

Une histoire partagée modélise une exécution distribuée d'un programme à l'aide d'un objet partagé.

Définition 5 (Histoires partagées et séquentielles)**.** *Une histoire partagée est composée d'un ensemble (fini ou infini) d'étapes étiquetées par les opérations de l'objet, et ordonné par l'ordre de processus, un ordre partiel tel que $e \mapsto e'$ si e et e' ont été exécutées par le même processus et dans ce même ordre.*

Une histoire séquentielle est une linéarisation *d'une histoire partagée H si elle contient les mêmes opérations que H, et que l'ordre d'apparition des opérations est compatible avec l'ordre de processus des événements étiquetés par les opérations.*

Un objet persistant comporte des opérations qui peuvent être invoquées par les processus et peuvent retourner une valeur. Ces transitions impliquent l'ensemble des états possibles de l'objet.

Définition 6 (Spécifications séquentielles)**.** *Un objet persistant propose des opérations qui peuvent être appelées par des processus et peuvent renvoyer une valeur. La* spécification séquentielle *d'un objet est l'ensemble de toutes les* histoires séquentielles *admises par l'objet, c'est-à-dire les séquences finies ou infinies d'invocations d'opérations et de réponses qui peuvent être produites lorsque toutes les opérations sur l'objet sont émises par un processus séquentiel unique.*

2.2 Description d'un modèle

Afin de définir précisément un modèle de calcul distribué, il faut pouvoir distinguer certains aspects dudit système. A la base d'un modèle, il faut définir le mode de fonctionnement des processus, les moyens et abstractions de communication, les paramètres de synchronie, les possibilités de panne, ainsi que le modèle d'arrivée définissant le nombre de participants au cours d'une exécution.

2.2.1 Modèles de communication

Les processus communiquent entre eux par différents moyens : dans le cas où les processus sont distribués géographiquement, la communication par passage de message est le moyen le plus intuitif de transmettre l'information. Si les processus sont sur un même site géographique (par exemple les systèmes *multi-thread*) il est possible de mettre en place une mémoire partagée à laquelle les processus accèdent pour communiquer.

Communication par passage de message

Les processus peuvent communiquer en envoyant et en recevant des messages via des canaux de communication. Les canaux de communication que nous considérons ici sont dits « fiables » : tous les messages envoyés sont finalement reçus par les processus corrects, et aucun message n'est dupliqué ou modifié. Des propriétés supplémentaires peuvent être ajoutées :

— par exemple un canal peut être *FIFO*, ce qui garantit que les messages envoyé par un processus à un autre, sont reçus par l'autre dans le même ordre dans lequel ils ont été envoyés.

— un canal peut être unidirectionnel ou bidirectionnel, dans le cas d'un canal unidirectionnel, un processus peut envoyer un message, mais le processus le recevant ne peut pas répondre via ce canal.

Chaque processus possède un ensemble de processus voisins, donc dans le cas d'un système par passage de message, le réseau peut être vu comme un graphe non dirigé (canaux bidirectionnels) ou dirigé (canaux unidirectionnels).

Selon le paramètre de synchronie choisi, il peut exister une borne sur les délais de communication (section 2.2.2).

Afin de permettre aux processus de communiquer avec l'entièreté du réseau, des primitives de diffusion peuvent être utilisées pour masquer les astuces de communication sous-jacentes.

Une vue d'ensemble des systèmes par passage de message est donnée dans [Ray13].

Communication par mémoire partagée

Dans un système de communication par mémoire partagée, pas de message direct entre les processus, mais une mémoire accessible concurremment par tous les processus. Il est possible pour chaque processus d'accéder aux différents registres et objets de la mémoire avec les opérations correspondantes en lecture ou en écriture.

Les processus ne sont pas limités dans le nombre de registres auxquels ils peuvent accéder pendant une exécution. Cependant, ils ne peuvent accéder qu'aux emplacements de mémoire alloués lors de la configuration du système, retournés par le mécanisme d'allocation, ou en suivant les références stockées (sous forme

de valeurs entières) dans un emplacement de mémoire accessible. C'est l'une des différences fondamentales entre un système par passage de message et un système en mémoire partagée.

Cependant, dans [ABD95], Un émulateur de système de mémoire partagée dans un système par passage de message robuste aux pannes est présenté : cela signifie que les résultats d'existence et d'impossibilité prouvés dans les modèles de mémoire partagée sont valables dans les modèles par passage de message sans perte de généralité (si moins de la moitié des participants peuvent tomber en panne). Le coût en nombre de message (n^2) impliqué par l'émulateur change aussi les résultats de complexité. Le modèle de mémoire partagée peut donc être considéré comme une abstraction de programmation d'un niveau supérieur dans la conception d'algorithmes distribués.

2.2.2 Modèles de synchronie

Processus synchrones : L'hypothèse de processus synchrones est liée à la capacité de synchroniser tous les processus selon un temps unique. Cela signifie que les processus et leurs communications peuvent être bornés en terme de temps d'exécution, ce qui permet de synchroniser une exécution en « rondes » bornée par le délai le plus long sur l'exécution et la communication parmi tous les processus. L'exécution de toutes les étapes de calcul effectuées lors d'une ronde r et la communication en découlant se terminent donc avant la ronde $r + 1$.

Processus asynchrones : A l'inverse du modèle de processus synchrones, lorsque l'on parle d'asynchronie, il n'y a pas de contrainte sur le processus qui prend chaque étape d'une exécution : un processus peut effectuer un nombre infini d'étapes consécutives, ou attendre un nombre non borné mais fini d'autres étapes de processus entre deux de ses propres étapes. De plus, il est possible qu'un processus cesse d'exécuter des étapes à un moment donné dans l'exécution même s'il n'a pas encore retourné de valeur pour son opération en cours, ce qui est assimilable à une panne.

En d'autres termes, l'asynchronie est représentée par l'absence de borne sur le temps d'exécution de chaque étape de calcul (ainsi que le temps de communication de l'information). Un processus attendant une mise à jour de la part d'un autre processus ne peut pas savoir, s'il n'a pas eu de « réponse », si c'est à cause de

problèmes de communication anormalement longue, ou parce que ledit processus est tombé en panne.

Une vue d'ensemble sur la différence entre la synchronie et l'asynchronie est présentée dans [Ray16].

2.2.3 Modèles de panne

Il convient de différencier les différentes hypothèses de pannes pour les processus.

Pannes franches : Les processus peuvent s'arrêter par panne franche : un processus *défectueux* s'exécute correctement jusqu'à ce qu'il *tombe en panne* ; à ce point, il cesse de fonctionner. Un processus qui ne tombe pas en panne pendant une exécution est appelé *correct*. Le nombre total de processus tombant en panne lors d'une exécution apporte des restrictions sur certains résultats de calculabilité. Des hypothèses sur les bornes sur le nombre total de processus pouvant tomber en pannes lors d'une exécution peuvent donc être faites. Dans [MR01] par exemple, l'hypothèse d'une borne $f < \frac{n}{2}$ (où n est le nombre de processus dans le système) est nécessaire pour proposer une solution au problème du consensus (présenté dans la section 2.4.1.).

Adversaires : Certains processus peuvent se comporter différemment de leur spécification et envoyer des données erronées de manière volontaire, dans le but de faire échouer les tâches en cours, ou de violer certaines propriétés pour certains objets. Ces adversaires sont appelés *adversaires byzantins* d'après le problème des généraux byzantins introduit par Lamport [LSP82]. Des solutions peuvent exister dans un système où de tels adversaires sont présents, mais souvent plus restrictives, comme le montre [**bracha1987asynchronous**] pour le problème du consensus par exemple.

2.2.4 Modèles d'arrivées

Cette dernière décennie, d'abord avec les systèmes pair-à-pair, puis avec le modèle multi-threading, l'hypothèse d'un système fermé avec un nombre fixe n de processus et où chaque processus connaît les identifiants de tous les processus est

devenue trop restrictive. D'où le modèle d'arrivée infinie introduit dans [MT03]. Nous pouvons distinguer trois types de modèles ouverts : le modèle d'arrivées bornées, le modèle d'arrivées finies, et le modèle d'arrivées infinies.

Plus précisément, les modèles de calcul se différencient par rapport aux restrictions du nombre d'arrivées de processus, comme présenté dans [GMT01]. Dans ces modèles, un certain nombre de processus peuvent tomber en panne (ou quitter, de la même manière que dans le modèle classique), mais de nouveaux processus peuvent également rejoindre le réseau lors d'une exécution. Lorsqu'un processus rejoint un tel système, il n'est pas connu des processus déjà en cours d'exécution. Quatre modèles sont distingués dans [Agu04] : le modèle classique (ou fermé), ainsi que trois nouveaux modèles ouverts.

Modèle classique : Le nombre n de processus est fixe et peut apparaître dans le code local du processus. Les processus peuvent toujours tomber en panne, mais aucun nouveau processus ne rejoindra le système au cours d'une exécution. C'est dans ce système que la plupart des résultats et preuves de possibilité ou d'impossibilité ont été démontrés.

Modèle d'arrivées bornées : Au plus N processus peuvent participer lors d'une exécution. La borne N n'est connue, par les processus, qu'au début de chaque exécution, mais peut varier d'une exécution à l'autre. Cela signifie que de nouveaux processus peuvent rejoindre l'exécution au cours du temps, mais le nombre total de participants ne dépassera jamais la borne N de cette exécution.

Modèle d'arrivées finies : Un nombre fini de processus participe à chaque exécution. Ce nombre étant non borné, il n'est pas connu des participants, mais il est garanti que les processus arrêteront de rejoindre le système à terme.

Modèle d'arrivées infinies : De nouveaux processus peuvent continuer d'arriver pendant toute l'exécution. Notons qu'à tout moment, le nombre de processus qui ont déjà rejoint le système est fini, mais peut croître à l'infini. Ce système est, du point de vue des processus, similaire au modèle d'arrivées finies, mais le fait qu'à tout moment un processus nouveau (et inconnu) puisse se rajouter au système apporte certains problèmes de concurrence et rend la synchronisation plus compliquée.

Remarque 1. *Un cinquième modèle, M_4, appelé* concurrence infinie*, a été introduit dans [Agu04], où une infinité de processus peuvent être présents dans le système et un nombre infini d'opérations peuvent avoir lieu dans n'importe quel intervalle de temps fini. Nous choisissons délibérément d'ignorer ce modèle car il pose un problème pour définir la linéarisation. Par exemple, supposons que, pour chaque $i \geq 1$, le processus p_i écrit la valeur i dans la variable x pendant l'intervalle $\left[1 - \frac{1}{2i}; 1 - \frac{1}{2i+1}\right]$; alors p_0 commence à lire x au moment 1. Il n'y a pas de « dernière valeur écrite » avant la lecture, donc la valeur de retour n'est pas bien définie. Restreindre la concurrence infinie à un sous-ensemble d'opérations non conflictuelles (*par exemple *lectures ou opérations sur différents objets) rendrait une concurrence infinie et une arrivée infinie équivalentes en terme de puissance de calcul comme nous pourrions utiliser facilement les conflits d'opérations concurrentes pour contrôler l'arrivée des processus.*

2.3 Cohérence et progression

Intuitivement, une « bonne » implémentation d'un objet partagé doit satisfaire deux types de propriétés : un critère de cohérence et une condition de progression, tout en respectant la spécification de l'objet. Le critère de cohérence spécifie la propriété de sûreté qui est la signification des résultats retournés, et la condition de progression spécifie les garanties sur la vivacité des opérations.

2.3.1 Critères de cohérence

Un critère de cohérence définit comment la concurrence affecte le comportement distribué d'un objet. Formellement, il identifie les histoires partagées qui sont admissibles pour une spécification séquentielle donnée. Il faut différencier la cohérence forte et la cohérence faible, pour laquelle nous autorisons certains passage incohérents dans une histoire.

Cohérence forte

Il s'agit de visualiser un objet partagé comme s'il s'agissait d'un seul objet physique partagé par tous les processus. Cela signifie que toutes les opérations sur l'objet, éventuellement simultanées ou entrelacées, semblent avoir été exécutées

atomiquement et séquentiellement. Cependant, ces implémentations sont souvent coûteuses, voire impossibles : le théorème CAP [GL02] indique que cette propriété est irréalisable dans la plupart des systèmes, car il est impossible de combiner une cohérence, une disponibilité et une tolérance aux partition élevées dans les systèmes asynchrones (Consistency, Availability, Partition tolerance). Les critères de cohérence faible ont été introduits pour surmonter ce problème.

Cohérence séquentielle : La cohérence séquentielle ([Lam79]) est le premier critère de cohérence forte introduit. Elle est définie comme suit.

Définition 7 (Cohérence séquentielle)**.** *Une histoire partagée est* Séquentiellement cohérente *si :*

- *Il existe une histoire séquentielle contenant toutes les opérations présentes dans l'histoire partagée (aucune opération n'a du être enlevée pour atteindre la cohérence).*
- *L'ordre de processus est respecté pour tous les processus.*
- *Cette histoire histoire séquentielle respecte la spécification de l'objet.*

Une histoire peut donc être séquentiellement cohérente même si certaines opérations sont ordonnées avant d'autres alors qu'elles leur sont ultérieures.

Linéarisabilité : La linéarisabilité [HW90] garantit que toutes les opérations renvoient la même valeur que si elles s'étaient produites instantanément à un instant donné dans le temps, appelé le *point de linéarisation*, entre leur invocation et leur réponse, éventuellement après avoir supprimé certaines opérations non terminées. Il faut donc considérer une histoire contenant comme évènements les invocations d'opérations, ainsi que les évènements de retour de ces opérations. Le point important de la linéarisabilité est le respect du temps réel par rapport à la cohérence séquentielle.

Définition 8 (Linéarisabilité)**.** *Une telle histoire est* linéarisable *si :*

- *l'histoire séquentielle obtenue par la projection des points de linéarisation des opérations respecte la spécification séquentielle de l'objet*
- *L'ordre des opérations dans l'histoire séquentielle respecte l'ordre temporel global : si le retour d'une opération a précède l'invocation d'une opération b dans le temps, alors a doit être placée avant b.*

Cohérence faible

Afin de palier la difficulté (voir l'impossibilité) d'implémenter la cohérence forte, la cohérence faible a été introduite. Elle se différencie de la cohérence forte par la possibilité de renvoyer des valeurs *incohérentes* entre les différents processus. Les deux critères qui nous intéressent dans cette thèse sont ceux de la *convergence* et de la *cohérence d'écriture*. Ces deux critères ont la particularité de converger vers un état unique cohérent après de potentielles périodes d'incohérence.

Convergence [Vog09] : Le critère de *convergence* implique qu'après l'arrêt des opérations de mise à jour, les différents réplicats convergeront, à terme, vers un état identique, nommé état de convergence [1].

Définition 9 (Convergence)**.** *Une histoire est convergente pour un objet O si, lorsque tous les processus cessent d'exécuter des opérations de mise à jour, ils finissent par converger vers un état commun. Formellement, une histoire H est convergente si elle se trouve dans l'un des deux cas suivants :*

- *Les processus n'arrêtent jamais d'effectuer des écritures, c'est-à-dire que H contient une infinité d'opérations de mise à jour.*
- *Il existe un état de convergence tel que toute opération effectuée après que cet état ait été atteint est équivalente à la même opération effectuée dans l'état de convergence.*

Il est utilisé en pratique dans de nombreuses applications à grande échelle telles que Dynamo, la base de donnée clé-valeur hautement disponible d'Amazon [DeC+07]. Il a été largement étudié et de nombreux algorithmes ont été proposés pour implémenter des objets partagés convergents.

Cohérence d'écriture [PMJ15] : La cohérence d'écriture renforce la convergence en déclarant que l'état de convergence doit pouvoir être obtenu dans une exécution séquentiellement cohérente. En d'autres termes, il peut être obtenu par un ordre séquentiel des opérations d'écriture de l'objet.

Définition 10 (Cohérence d'écriture)**.** *Une histoire est cohérente en écriture (ou update cohérente) pour un objet O si, lorsque tous les processus cessent d'exécuter*

1. Remarquons que l'état de convergence n'est pas spécifié.

des opérations de mise à jour, ils finissent par converger vers un état résultant d'une linéarisation des opérations d'écriture. Formellement, une histoire H est cohérente en écriture si elle se trouve dans l'un des deux cas suivants :

- *Les processus n'arrêtent jamais d'effectuer des écritures, c'est-à-dire que H contient une infinité d'opérations de mise à jour.*
- *Il est possible d'omettre un nombre fini d'opérations de requête de telle sorte que l'histoire en résultant ait une linéarisation dans la spécification séquentielle de O.*

D'un point de vue calculabilité, et de manière similaire à la convergence, il est possible d'implémenter n'importe quel objet avec le critère de cohérence d'écriture [PMJ15 ; PMJ16].

2.3.2 Conditions de progression

L'utilisation de verrous dans l'implémentation peut provoquer un blocage dans un système où les processus peuvent tomber en panne, en effet un processus peut détenir un verrou et tomber en panne, ne libérant ainsi pas ledit verrou. D'autres processus étant en attente de ce verrou seraient donc bloqués indéfiniment. Certaines conditions de progressions telles que la *deadlock-freedom* [RD87] et la *starvation-freedom* [HS08] sont appelées conditions bloquantes, c'est à dire qu'elles sont rendues caduques dans le cas où ne serait-ce qu'un seul processus tomberait en panne. A l'inverse, les conditions telles que la *wait-freedom* [Her91] la *lock-freedom* [HW90], et l'*obstruction-freedom* [HLM03] ne dépendent pas du nombre de pannes dans le système (à part si tous les processus tombent en panne, dans ce cas aucune progression n'est possible évidemment). Ces conditions de progressions sont appelées non bloquantes.

Les conditions de progression sont globales ou locales : alors que la *wait-freedom* garantit que chaque opération se termine après un temps fini, la *lock-freedom* garantit que, si le calcul s'exécute assez longtemps, au moins un processus progresse (cela peut conduire à une situation de famine pour certains processus). La *wait-freedom* est donc plus forte que la *lock-freedom* [DW15] : alors que la *lock-freedom* est une condition à l'échelle du système (globale), la *wait-freedom* est une condition propre au processus (locale). De même, alors que la *deadlock-freedom* est une garantie à l'échelle du système, la *starvation-freedom* est une condition

locale.

Conditions de progression bloquantes

Dans le cas d'un système asynchrone, ces conditions spécifient les conditions de progression en cas d'absence de panne franche lors de l'exécution.

Starvation-freedom **[HS08] :** La *Starvation-freedom* stipule que s'il n'y a aucune panne lors d'une exécution, alors tous les processus progressent et terminent.

Deadlock-freedom **[RD87] :** La *Deadlock-freedom* stipule que s'il n'y a aucune panne lors d'une exécution, alors au moins un processus dans le système progresse.

Conditions de progression non bloquantes

Dans un système asynchrone où des pannes franches peuvent survenir, il faut que les conditions de progression soient non bloquantes, et ne dépendent donc pas de la présence ou non de pannes.

Wait-freedom **[Her91] :** La *Wait-freedom* est la condition de progression la plus forte, elle signifie qu'aucun processus ne peut être bloqué pendant un temps infini, et garantit donc la terminaison de toute opération commencée. C'est généralement la condition de progression désirée pour l'implémentation d'un objet partagé. C'est une condition de progression locale.

Définition 11 (Wait-freedom)**.** *Un algorithme A est* wait-free *si, pour toute exécution α de A, aucune opération ne prend un nombre infini d'étapes dans α.*

Lock-freedom **[HW90] :** La *Lock-freedom* signifie qu'il y a à tout instant un processus qui progresse. c'est une condition globale qui garantit que le système dans sa globalité progresse, même si certains processus peuvent être bloqués indéfiniment.

Définition 12 (Lock-freedom)**.** *Un algorithme A est* lock-free *si, pour toute exécution α de A, au moins un processus n'a aucune opération prennant un nombre infini d'étapes dans α*

***Obstruction-freedom* :** L'*Obstruction-freedom* signifie que si un processus s'exécute seul pendant assez longtemps (sans que les processus pouvant faire *obstruction* à sa progression n'interfèrent) il est garanti de progresser. Cette condition est plus faible que la *lock-freedom* : en effet, même si elle garantit toujours une tolérance aux pannes, il n'est pas garanti que le système progresse en présence de concurrence.

2.4 Universalité et constructions universelles

La question de l'universalité a toujours été centrale dans tous les domaines scientifiques. Que peut-on faire avec un outil et un contexte donnés, et surtout qu'est ce qui ne peut pas être fait avec un tel outil. En informatique séquentielle, l'universalité est représentée par une machine de Turing qui peut calculer tout ce qui est calculable. Un problème central du calcul distribué est l'accord, résumé par le problème du consensus. Dans le contexte des systèmes distribués, nous savons depuis 1985 et le fameux résultat d'impossibilité FLP que le problème du consensus n'a pas de solution déterministe dans un système distribué où même un seul processus peut tomber en panne [FLP85]. Cette impossibilité n'est pas due à la puissance de calcul des processus individuels, mais plutôt à la difficulté de coordination entre les différents processus qui composent le système distribué. Les problèmes de coordination et d'accord sont donc au cœur de la calculabilité dans les systèmes distribués [HRR13].

On dit qu'un modèle de calcul distribué M est *Wait-free universel* (ou simplement universel) si n'importe quel objet avec une spécification séquentielle peut être implémenté de manière *wait-free* et linéarisable dans M. Par extension, un objet O est dit universel dans M si $M[O]$ (le système M enrichi de l'objet O) est universel. Afin de classer les objets selon leur puissance de synchronisation, Herlihy a proposé un classement appelé *Hiérarchie wait-free* [Her91], qui trie les objets par numéro de consensus croissant.

Définition 13 (Numéro de consensus)**.** *Soit O un objet. On dit que O a le numéro de consensus $n \in \mathbb{N}$ si O est universel dans un système à n processus mais pas dans un système à $n+1$ processus. O a un numéro de consensus infini (∞) s'il est universel pour tout n.*

2.4.1 Consensus

Définition 14 (consensus)**.** *Le problème du consensus vise à atteindre un accord entre plusieurs processus proposant chacun une valeur, et décidant tous une unique valeur. Il vérifie trois propriétés :*

Validité Toute valeur décidée fait partie des valeurs proposées.

Accord Deux processus ne peuvent pas décider deux valeurs différentes.

Terminaison Si un processus est correct, il finit par décider une valeur.

Maurice Herlihy a prouvé dans [Her88] que le consensus est universel dans les systèmes distribués classiques composés d'un ensemble de n processus. A savoir, tout objet ayant une spécification séquentielle a une implémentation *wait-free* et linéarisable utilisant uniquement des registres de lecture / écriture (emplacements de mémoire) et un certain nombre d'objets de type consensus. Cela fait du consensus un objet de numéro de consensus infini.

Une variante du consensus appelée *consensus binaire* (parfois appelée consensus booléen) propose un processus de décision parmi deux valeurs possibles. Le consensus « classique » est donc aussi appelé *consensus multi-valué.*

Définition 15 (Consensus binaire)**.** *Le problème du consensus binaire vise à atteindre un accord entre plusieurs processus proposant chacun une valeur parmi deux valeurs possibles, et décidant tous une unique valeur. Il vérifie les trois même propriétés que le consensus :*

Validité Toute valeur décidée fait partie des valeurs proposées.

Accord Deux processus ne peuvent pas décider deux valeurs différentes.

Terminaison Si un processus est correct, il finit par décider une valeur.

Il est possible d'implémenter le consensus multi-valué à partir du consensus binaire comme démontré dans [MRT00], ce qui rend ces objets équivalents dans la hiérarchie de Herlihy.

Certaines instructions matérielles spécifiques atomiques permettent d'implémenter intuitivement le consensus.

Définition 16 (*Compare&swap*)**.** *L'opération matérielle spécifique compare&swap fonctionne de la manière suivante : l'appel de compare&swap sur X, $X.compare\&swap(attente, v)$, vérifie si la valeur contenue dans X est égale à la valeur attente, et si oui, remplace la valeur contenue dans X par la valeur v et renvoie* **true**. *L'opération*

renvoie **false** *sinon. Tout cela se déroule de manière atomique. Pour implémenter le consensus sur une variable X initialisée à $\perp$ (valeur vide), il suffit d'appeler X.compare&swap$(\perp, v)$, puis de lire la valeur de X. Une seule valeur sera écrite, et puisque l'opération est atomique, tous les autres processus obtenant* **false** *lors de leur invocation de compare&swap n'auront donc pas pu modifier la valeur contenue dans X, et le consensus est ainsi atteint avec pour valeur décidée l'unique valeur écrite dans X.*

Le consensus dans les systèmes ouverts Le consensus fut à la base étudié d'abord dans des systèmes synchrones ([PSL80]), puis dans les systèmes asynchrones ([FLP85]). Des solutions ont enfin été proposées dans le modèle d'arrivées infinies :

— Dans [ASS02], pour répondre au problème du consensus binaire, un système de votes et de poids permet aux processus de décider même en présence d'adversaires. Plus un participant arrive tard dans le système, moins son vote a de poids dans la décision finale.

— Dans [CM05], une variante du protocole Paxos ([Lam+01]) est étendue à un nombre infini de processus pour résoudre le problème du consensus en mémoire partagée.

— Dans [MT03], le consensus pour une infinité de processus est atteint à l'aide de consensus pour un nombre fini de processus, et d'objets *test&set*. Pour cela un nombre d'objets *test&set* est utilisé pour permettre à un nombre t de processus de participer à un consensus pour t participants.

Définition 17 (*Test&set*)**.** *L'opération matérielle spécifique Test&set fonctionne de la manière suivante : l'appel de test sur un objet Test&set X (initialisé à* **false***), X.test&set(), vérifie que la valeur contenue dans X est* **false***, et si oui, remplace la valeur de X par* **true** *et renvoie* **true***, et renvoie* **false** *sinon. Cette opération étant atomique, seul un processus peut obtenir le résultat* **true***. Cette opération matérielle a pour numéro de consensus* 2*.*

2.4.2 Objets au numéro de consensus infini dans la hiérarchie de Herlihy

Dans la hiérarchie de Herlihy, tous les objets ayant un numéro de consensus infini ne sont pas distinguables au vu d'un système classique à n processus, mais sont-ils pour autant équivalents dans les modèles ouverts ?

Le consensus a été utilisé comme base de raisonnement sur la calculabilité dans ces modèles [AMW11], et déjà une séparation en terme de calculabilité entre les objets ayant un numéro de consensus infini est apparue : il existe un objet de numéro de consensus infini qui, bien qu'il soit capable de réaliser un consensus pour n'importe quel nombre de processus n (même si n est inconnu à l'avance), ne peut pas résoudre le consensus face à une concurrence illimitée. L'objet en question est l'*Iterator Stack* (ou `IStack`, son fonctionnement est montré dans la section 3.1), et il est démontré que l'*Iterator Stack* permet d'implémenter le consensus dans un modèle d'arrivées finies, mais pas dans le modèle d'arrivées infinies moins restrictif. Cela marque une séparation entre les modèles d'arrivées finies et infinies dans la classe des objets de numéro de consensus infini.

2.4.3 Constructions universelles

Pour prouver l'universalité du consensus, Herlihy a introduit la notion de construction universelle [2]. Il s'agit d'un algorithme générique qui, étant donné une spécification séquentielle de tout objet dont les opérations sont totales [3], fournit une implémentation concurrente de cet objet.

Constructions universelles *wait-free* et linéarisable

Construction universelle *wait-free*, linéarisable, sur base d'objets LL/SC (Linked Load / Store Conditionnal, avec mécanisme d'*entraide* Proposée dans [Ray17], Cette construction universelle en mémoire partagée se base sur des objets LL/SC, et un objet *Collect*. L'objet LL/SC est composé de trois opérations : soit X un emplacement mémoire,

— $X.LL$ renvoie la valeur de X.

2. Une petite visite guidée sur les constructions universelles se trouve dans [Ray17].

3. Cela signifie que toute opération sur l'objet peut être appelée et l'appel retourne quel que soit l'état de l'objet.

— si un processus p invoque $X.SC(v)$, alors X est mis à v si aucun autre processus n'a modifié X depuis la dernière invocation de $X.LL$ par p, et renvoie **true**, **false** sinon.
— si un processus p invoque $X.LL$, alors si aucun autre processus n'a pu invoquer $X.SC(v)$ avec succès depuis le dernier appel de $X.LL$ par p, cela renvoit **true**, **false** sinon.

Cet objet permet à un processus de modifier une valeur sans que celle ci ait été modifiée depuis sa dernière lecture, et ainsi d'éviter des incohérences dans les données : par exemple si deux processus essaient d'incrémenter un compteur concurremment, si les deux processus lisent une valeur x et essaient d'écrire $x+1$, alors le résultat final sera $x+1$ au lieu de $x+2$, l'utilisation d'un LL/SC évite ce problème.

L'objet *Collect* est un vecteur de valeurs dont les cases sont détenues par chacun des processus, qui leur permet de mettre à jour leur valeur, ou de « collecter » la totalité des valeurs du vecteur, de manière non atomique.

La construction universelle proposée dans [Ray17] fonctionne de la manière suivante :

1. lorsqu'un processus invoque une opération o, il l'insère d'abord dans un objet *Collect*
2. dans un second temps, le processus collecte toutes les valeurs de l'objet *Collect* (donc l'ensemble des opérations que les processus essaient d'invoquer).
3. il essaie ensuite d'ajouter toutes les opérations collectées précédemment dans un objet LL/SC.

Ceci correspond à un mécanisme d'*entraide* formalisé dans [CPT15]. Ce mécanisme nécessite que tout processus, avant de terminer son fonctionnement, aide les processus ayant des opérations en attente, ceci afin d'atteindre la *wait-freedom*.

D'autres constructions universelles basées sur un objet *Collect* et le même principe d'entraide, mais avec d'autres objets que le LL/SC, comme le consensus et l'instruction spécifique Compare&swap, sont présentées dans [Ray17].

Construction universelle *wait-free*, linéarisable, sur base d'opérations *Fetch&Cons*, sans entraide Une opération *fetch&cons* sur une liste l, avec en argument une valeur v, exécute atomiquement les deux opérations suivantes :

1. Concaténation de v à la tête de la liste l.
2. Retour de l'entièreté de la liste l qui précède v (sans v).

Une construction universelle utilisant pour base cette opération est proposée dans [Her88]. Le fonctionnement est assez simple : un processus invoquant une opération o fait un appel à $fetch\&cons(v, l)$. Étant atomique, cette opération *fetch&cons* a un point de linéarisation (cela garantit la linéarisabilité : l'ordre total des opérations est l'ordre des points de linéarisation). Le processus exécute ensuite localement la liste des opérations retournée depuis l'état initial de l'objet, pour connaître l'état dans lequel évaluer son opération, puis exécute son opération.

Dans [CPT15], en supposant qu'une opération sur les liste *fetch&cons* soit disponible et fonctionne sans mécanisme d'entraide [4], il est montré que la propriété d'absence d'entraide (*help-free* dans la littérature) reste vraie pour cette construction universelle, étant donné qu'aucun mécanisme d'entraide supplémentaire n'est ajouté.

Constructions universelles *wait-free* et faiblement cohérentes

Les constructions universelles existent aussi en cohérence faible : [PMJ15] introduit une construction universelle *update* cohérente dans un système par passage de messages.

Construction universelle *wait-free*, *update* cohérente Cette construction universelle construit un ordre total sur les opérations de mises à jour sur lequel tous les participants sont d'accord *a priori*, puis réécrit l'historique *a posteriori* quand de nouveaux messages arrivent (Cet ordre est construit à partir d'une horloge de Lamport [Lam78]). Tous les processus atteignent donc finalement l'état correspondant à l'histoire séquentielle commune. Lorsqu'un processus appelle une opération, une mise à jour est émise localement, le processus l'horodate et informe tous les autres processus en diffusant un message pour tous les autres processus. Tous les processus seront donc éventuellement informés de toutes les mises à jour, ce qui assure la convergence des différents processus. Quand un processus veut effectuer une lecture sur l'objet, il rejoue localement toute la liste des événements

4. [CPT15] prouve cependant que l'implémentation d'un objet *fetch&cons* est impossible sans mécanisme d'entraide, il faut donc que le système fournisse une telle opération (via une instruction matérielle par exemple).

de mises à jour dont il a conscience à partir de l'état initial, puis il exécute la requête sur l'état qu'il vient d'obtenir. Les messages arrivant « à terme », il ne peut exister qu'un nombre fini de lectures sur des états incohérents, ce qui garantit bien la cohérence d'écriture (si les processus n'arrêtent jamais d'invoquer des écritures, le critère de la cohérence d'écriture est respecté par définition).

Cette construction universelle *update* cohérente a pour avantage qu'un seul message est envoyé par opération de mise à jour de l'objet.

2.5 Sous-modèles

En gardant les mêmes hypothèses vis à vis de tous les critères d'un modèle mentionnés (communication, synchronie, pannes, arrivées), il est possible de restreindre les algorithmes à certaines formes précises pour gagner en simplicité et en efficacité : Cela donne lieu à des sous-modèles. Le cas que nous allons aborder est celui du modèle opérationnel pour la cohérence faible en système par passage de message, introduit pour l'implémentation des CRDT.

Complexités Les critères de complexités que nous pouvons étudier sont les suivants :

- La complexité en mémoire locale (aussi appelée complexité spatiale), c'est à dire la quantité de données stockée par chaque processus au long d'une exécution. Cette complexité dépend généralement du nombre de processus présents lors d'une exécution, et/ou du nombre d'opérations effectuées dans ladite exécution.
- La complexité en nombre de messages envoyés par opération (aussi appelée complexité en communication). Cette complexité dépend généralement du nombre de processus présents lors d'une exécution.
- La complexité en taille des messages envoyés. Cette complexité dépend généralement du nombre de processus présents lors d'une exécution, et/ou du nombre d'opérations effectuées dans ladite exécution.

CRDT et modèle opérationnel Les *types de données répliquées sans conflits* (ou CRDT) [Sha+11] constituent une famille d'objets spécialement conçus pour répondre au critère de convergence. Ceux-ci sont basés sur un théorème énonçant

l'équivalence entre deux types d'objets : les types de données répliquées commutatives (CmRDT), dans lesquelles toutes les opérations de mise à jour commutent, et les types de données répliquées convergentes (CvRDT), dont les états forment un treillis. Par exemple, le G-set (Grow-Only set, ensemble strictement croissant) fournit deux opérations différentes : une opération de mise à jour qui insère un élément et une opération de requête qui lit si un élément spécifique se trouve dans l'ensemble. Sur le point de vue CmRDT, insérer x et insérer y commutent. Du point de vue CvRDT, l'inclusion de l'ensemble est un ordre formé par le treillis sur les états de l'ensemble. Le *modèle opérationnel* a été proposé pour abstraire la mise en œuvre des CRDT. Dans le modèle opérationnel, chaque processus conserve un état local sur lequel les opérations sont effectuées. Une opération de mise à jour est divisée en deux facettes. Tout d'abord, l'opération de mise à jour est préparée localement par le processus d'où l'opération de mise à jour est émise, puis un message est diffusé pour informer tous les autres processus. Deuxièmement, l'état local de chaque processus est mis à jour à la réception du message de mise à jour. Grâce à la commutativité, tous les réplicats convergent vers le même état lorsqu'aucune opération de mise à jour n'est en cours.

Comme un seul message est diffusé par opération de mise à jour, les algorithmes du modèle opérationnel sont, par conception, optimaux en termes de nombre de messages utilisés. La quantité de métadonnées qui doivent être stockées par chaque processus pour garantir la convergence, ou la taille des messages à envoyer est plus problématique et a été largement étudiée pour plusieurs objets, notamment les ensembles (*OR-set* : $O(n\,log(m))$ en complexité spatiale, avec n le nombre de processus, et m le nombre d'opérations de mise à jour effectuées), les compteurs ($O(n)$ en complexité spatiale) et les registres ($O(m)$ en complexité spatiale) dans [Bur+14], les bases de données (borne inférieure sur la taille des messages en $O(n\,log(m))$) dans [AEM17] et les éditeurs collaboratifs ($O(|l| + D)$ où l est la liste représentant l'éditeur collaboratif, et D le nombre de suppressions d'éléments de la liste) dans [Att+16]. Malgré le fait que les algorithmes du modèle opérationnel soient naturellement tolérants aux partitions (En effet, dans ce modèle, l'état de l'objet dépend uniquement de l'ensemble des messages reçus. Si l'on dispose d'un mécanisme de diffusion fiable, les ensembles de messages reçus dans les différentes partitions convergent lorsque ces partitions sont résolues) et minimisent le nombre de messages nécessaires à leur mise en œuvre, le modèle

opérationnel impose des limitations sur la forme de ses algorithmes admissibles. Il n'est par exemple pas autorisé d'accuser réception ou de retransmettre des messages, d'exécuter des étapes locales sans réception de message, ou de propager des informations lors des opérations de lecture. Cela empêche les algorithmes d'utiliser des techniques plus avancées telles que les schémas de message utilisés par les algorithmes de *checkpointing* [Bal+95 ; RLT78] [5]. De tels algorithmes sont généralement étudiés dans le *modèle wait-free par passage de message* ou plus simplement modèle *wait-free*, dans lequel les processus asynchrones communiquent en envoyant et en recevant des messages. N'importe quel nombre de processus peut tomber en panne, ce qui capture également la tolérance aux partitions car il est impossible pour un processus d'attendre un accusé de réception de tout autre processus car tous les autres processus peuvent être tombés en panne.

Le modèle opérationnel et les CRDT sont naturellement adaptés aux modèles ouverts, car la convergence est atteinte lorsque tous les messages sont reçus par l'ensemble des participants.

5. Le principe du *checkpointing* est de sauvegarder (ou « fixer ») l'état local des processus au fur et à mesure de l'exécution. De tels état (ou *checkpoints*) doivent être cohérents, ce qui permet d'« oublier » certaines opérations menant à cet état par exemple, et de gagner ainsi en complexité spatiale.

Chapitre 3

Description des modèles utilisés

Dans ce chapitre, nous décrirons les différents modèles utilisés dans les parties suivantes, du mode de communication au modèle ouvert utilisé.

Dans la seconde partie de la thèse, la seule différence entre les modèles utilisés et celui ci est la communication par mémoire partagée (puis pour l'intérêt de la hiérarchie, le modèle d'arrivée). Dans la troisième partie de la thèse, l'intérêt étant la complexité, le modèle est le même qu'O'browser, mais nous étudions aussi le sous modèle qu'est le modèle opérationnel.

3.1 Seconde partie : communication par mémoire partagée

Les modèles que nous utiliserons dans la seconde partie ont pour base les même caractéristiques :

— Une communication par mémoire partagée.
— Un système asynchrone.
— Tous les processus sauf un peuvent tomber en panne.

Pour les modèles d'arrivées, chaque paramètre donnera lieu à un nouveau modèle.

Nous décrirons donc en détail le fonctionnement de la mémoire et les objets que nous utiliserons dans les chapitres 4 et 5, puis nous nommerons tous les modèles que nous utiliserons.

Communication entre processus : Les processus communiquent en lisant et en écrivant une mémoire composée d'un nombre infini de registres illimités [1].

Dans le chapitre 4, les processus ont accès à un mécanisme d'allocation qui ne peut renvoyer qu'un nombre non borné, mais fini, d'emplacements de mémoire à la fois et qui n'alloue jamais deux fois le même emplacement de mémoire. Dans le chapitre 5, afin d'ajouter une dimension à l'extension de la hiérarchie de Herlihy, et afin d'étudier l'effet de l'accès à une allocation mémoire infinie sur la puissance de synchronisation des objets, les processus ont accès à un mécanisme d'allocation d'un nombre infini d'emplacements de mémoire.

Les processus sont anonymes, en ce sens qu'ils ne connaissent pas leur identifiant unique : cette hypothèse modélise le fait qu'il serait déraisonnablement coûteux de baser un algorithme sur des identifiants illimités, en particulier lorsque l'attribution des identifiants n'est pas basée sur l'ordre d'arrivée des processus dans le système. De plus, en supposant des processus anonymes, le résultat n'en est pas restreint.

Composition de la mémoire : La mémoire est composée de deux types de registres : les registres immuables et les registres de lecture/écriture.

Un registre immuable x est initialisé avec une valeur fixe et ne peut pas être modifié ultérieurement. Il fournit une opération de lecture, simplement notée x dans les algorithmes, qui renvoie la valeur interne du registre.

Un registre de lecture/écriture x fournit deux opérations : une opération de lecture $x.\texttt{read}()$ qui retourne la valeur interne du registre et une opération d'écriture $x.\texttt{write}(v)$ qui remplace la valeur actuelle de x par v.

Objets de synchronisation. Afin d'améliorer leur calculabilité, les différents modèles de calcul peuvent être enrichis en donnant accès à des objets atomiques partagés plus évolués, qui sont indiqués entre crochets dans le nom du modèle et appelés objets partagés enrichissants. Par exemple, $M_3[\texttt{cons}\langle\mathbb{N}\rangle]$ désigne le modèle d'arrivée infinie où autant d'objets consensus que nécessaire sont disponibles. Chaque objet partagé enrichissant est atomique dans le sens où les différentes exécutions des appels sur ses opérations sont totalement ordonnées.

1. Les adresses mémoire d'une mémoire infinie étant illimitées, cette hypothèse est nécessaire pour stocker des références. [Ell+16].

Registres fenêtrés. Un registre fenêtré de taille k [PMJ16], noté k-`WReg`, a deux opérations : une opération d'écriture `write`(v), qui prend un argument entier $v \in \mathbb{N}$ et ne renvoie aucune valeur, et une opération de lecture `read`() qui renvoie la liste ordonnée des k dernières valeurs écrites. Certaines valeurs peuvent être remplacées par les valeurs par défaut $\perp$ si moins de k écritures précèdent la lecture. Notez que k-`WReg` généralise le registre de lecture/écriture classique qui correspond au cas spécial $k = 1$. `WReg` est la généralisation dans laquelle les registres fenêtrés de toutes tailles sont accessibles : la taille k est passée en argument du constructeur.

Piles d'itérateurs. La pile d'itérateurs `IStack` [AMW11], fournit une opération d'écriture `isWrite`() et une opération de lecture `isRead`(). Intuitivement, `isWrite`(v) ajoute la valeur v au début d'une pile et renvoie une référence i à un nouvel itérateur, et `isRead`(i) incrémente l'itérateur i et renvoie la valeur vers laquelle il pointe. Plus précisément, `isWrite`(v) prend comme argument une valeur écrite $v \in \mathbb{N}$ et renvoie la prochaine valeur entière dans une séquence $0, 1, \ldots$ Pour un $i \in \mathbb{N}$ donné, l'invocation de la $k^{\text{ème}}$ opération `isRead`(i) renvoie la $k^{\text{éme}}$ valeur jamais écrite si `isWrite` a été invoquée au moins $\max(i + 1, k)$ fois, et $\perp$ sinon.

Objets consensus. Un objet consensus, noté $\mathsf{cons}\langle T\rangle$, fournit une opération `propose`(v) qui prend un argument $v \in T$ et retourne la plus ancienne valeur proposée, c'est-à-dire que le premier processus qui invoque l'opération obtient sa propre valeur et toutes les invocations suivantes retournent cette même valeur, appelée valeur décidée. On distingue le consensus binaire $\mathsf{cons}\langle\mathbb{B}\rangle$ dans lequel seules les valeurs **true** et **false** peuvent être proposées. Enfin, $\mathsf{cons}^n\langle T\rangle$ désigne le consensus à n processus, dans lequel les propriétés précédemment déclarées ne sont vérifiées que par les n premières invocations de `propose`, et les valeurs retournées suivantes sont laissées non spécifiées.

Dans le chapitre 4, nous utiliserons la définition suivante pour le consensus : un objet consensus x fournit deux opérations : une opération de lecture similaire à l'opération de lecture sur un registre immuable, et une opération x.`propose`(v) qui vérifie atomiquement si la valeur actuelle est $\perp$, puis fixe la valeur de x à v si c'est le cas, et renvoie finalement la valeur de x. En d'autres termes, la première

valeur proposée est écrite sur x, qui devient immuable à partir de ce moment [2].

Registre *compare&swap*. Un registre CAS x fournit deux opérations : une opération de lecture sur x fonctionnant comme l'opération de lecture du registre immuable, et une opération de compare&swap $x.\texttt{CAS}(expect, update)$ qui vérifie atomiquement si la valeur actuelle de x est *expect*, la change en *update* et renvoie **true** si c'est le cas. Si $x \neq expect$, alors $x.\texttt{CAS}(expect, update)$ renvoie **false** sans changer la valeur de x.

Modèles d'arrivées : Chaque instance d'un modèle de mémoire et d'un modèle d'arrivée donne un modèle. Pour la mémoire « de base », nous prenons seulement les registres immuables et les registres de lecture/écriture. Pour le mécanisme d'allocation de mémoire fini, cela nous donne les 4 modèles suivants. Etant donné que le seul élément permettant de distinguer ces modèles est leur modèle d'arrivée, nous les appellerons, par abus de language, comme cela par la suite :

— Le modèle classique, M_1^n.
— Le modèle d'arrivées bornées, M_1.
— Le modèle d'arrivées finies, M_2.
— Le modèle d'arrivées infinies, M_3.

Dans le chapitre 5, lorsque des objets de synchronisation suffisamment forts ne sont pas disponibles, il peut être nécessaire d'activer le mécanisme d'allocation permettant d'allouer et d'initialiser un nombre infini d'emplacements de mémoire à la fois. Lorsqu'un système permet une telle allocation, il est censé fournir une allocation infinie. Cela définit quatre autres modèles de système d'arrivée MA_1^n, MA_1, MA_2 et MA_3 qui représentent les quatre modèles susmentionnés enrichis d'un mécanisme d'allocation de mémoire infini.

De plus, comme certains objets ne peuvent pas être implémentés en utilisant uniquement des registres de lecture/écriture, un système peut être enrichi avec des objets de synchronisation tels que des objets consensus, des piles d'itérateurs, etc. Ce qui donne lieu à de nouveaux modèles (notés $M_1[Obj]$ etc ...).

Ces modèles permettent détendre la définition du numéro de consensus de la

2. Cette définition est proche du *sticky bit* et n'est pas exactement la même que celle communément acceptée du consensus. Cependant, il est facile de le mettre en œuvre en utilisant un processus de consensus et un registre de lecture/écriture, initialisé à $\perp$ et écrit lorsque le consensus est décidé. Nous utilisons cette définition dans un souci de clarté des algorithmes proposés.

manière suivante :

Définition 18 (Numéro de consensus)**.** *Soit O un objet. On dit que O a le numéro de consensus $n \in \mathbb{N}$ si O est universel dans un système à n processus mais pas dans un système à $n+1$ processus, et que O a un numéro de consensus ∞_x^y, pour $x, y \in \{1, 2, 3\}$ s'il vérifie les deux conditions suivantes :*
1) $M_x[O]$ est universel et, si $x \leq 2$, alors $M_{x+1}[O]$ n'est pas universel, et
2) $MA_y[O]$ est universel et, si $y \leq 2$, alors $MA_{y+1}[O]$ n'est pas universel.

3.2 Troisième partie : communication par passage de message

Les bases des modèles utilisés dans la troisième partie sont les suivantes :

— Une communication par passage de message.
— Un système asynchrone.
— Tous les processus sauf un peuvent tomber en panne.
— Une arrivée potentiellement infinie de processus.

Ce modèle, similaire à celui d'O'browser, est appelé modèle *wait-free* par passage de message. Nous l'appellerons dans la suite modèle *wait-free*.

Pour la communication nous détaillerons les abstractions de communication que nous utiliserons. Les deux modèles utilisés dans cette partie étant liés (l'un est un sous modèle de l'autre en terme de restriction algorithmique), le modèle de calcul distribué sous-jacent est fondamentalement le même.

Communication entre processus : Les processus peuvent communiquer en envoyant et en recevant des messages. Les canaux de communication sont fiables : tous les messages envoyés sont finalement reçus par les processus corrects. Cependant, les canaux sont asynchrones, dans le sens où il n'y a pas de limite sur le temps qu'il faut à un message pour être délivré. Nous supposons que tous les messages envoyés peuvent être identifiés de manière unique.

Nous supposons que les processus ont accès à une abstraction de *diffusion causale* qui leur fournit une opération **broadcast** m et un événement **receive** m, où m est un message, en respectant les propriétés suivantes.

— Validité : si un processus délivre un message m, alors m a été diffusé par un processus.
— Uniformité : si un processus délivre un message m, alors tous les processus corrects délivrent m.
— Terminaison : si un processus correct p_i tente de diffuser m, son appel à l'opération de diffusion termine et p_i délivre m à terme.
— Livraison causale : si un processus délivre un message m puis diffuse un message m', alors tous les processus délivrant m' ont déjà délivré m.

Notez que la diffusion causale peut être implémentée dans le modèle *wait-free* [RST91] sans puissance de calcul supplémentaire. Cependant, cette implémentation a un coût en mémoire locale. Nous choisissons d'inclure directement la primitive dans le modèle pour isoler la complexité nécessaire au maintien de la cohérence des objets partagés de la complexité nécessaire à assurer la causalité, et donc réduire le bruit des résultats de complexité que nous obtenons.

Spécification séquentielle Dans un souci de clarté et pour se conformer aux hypothèses nécessaires pour spécifier les CRDT, nous ne considérerons que deux types d'opérations dans le chapitre 6 : les opérations de mise à jour (ou écriture) qui modifient l'état local de l'objet mais ne renvoient pas de valeur, et les opérations de requête (ou lecture) qui renvoient une valeur en fonction de l'état de l'objet mais ne le modifient pas.

Modèle opérationnel Le modèle opérationnel est composé de réplicats r_1, ..., r_n, etc... Tous les réplicats exécutent le même algorithme selon un format très spécifique, qui définit le modèle, et qui est défini par la suite.

Chaque réplicat maintient un état local, appelé $\mathsf{payload}$, et peut interagir avec le système en invoquant deux types d'opérations. Tout d'abord, les opérations de requête q retournent une valeur $q(\mathsf{payload})$ qui est calculée localement en fonction de l'état local du réplicat. Deuxièmement, les opérations de mise à jour u sont séparées en une fonction $\texttt{prepare}_u$ et une fonction $\texttt{effect}_u$. La fonction $\texttt{prepare}_u$ calcule localement une information data basée sur la fonction de mise à jour et l'état local. Ensuite, $\texttt{effect}_u(\mathsf{data}, \mathsf{payload})$ est appliqué de manière asynchrone sur l'état local de tous les réplicats. Il est nécessaire que toutes les fonctions $\texttt{effect}_u$ commutent, afin que tous les réplicats puissent finalement converger

vers un état commun.

Un processus rejoignant le système n'a ainsi qu'à effectuer l'ensemble des fonctions $\texttt{effect}_u$ et de les appliquer à l'état initial, ce qui est équivalent à un processus qui aurait été victime de partition jusqu'à cet instant.

Les algorithmes du modèle opérationnel peuvent être considérés comme un cas particulier d'algorithmes du modèle *wait-free*. Plus précisément, l'algorithme 1 est une injection canonique qui lie tout algorithme A dans le modèle opérationnel à un algorithme $op2wf(A)$ dans le modèle *wait-free*. Chaque réplicat du modèle opérationnel est intégré sur un processus dans le modèle *wait-free*, qui maintient les données payload de A comme état local. Lorsqu'un processus effectue une opération de requête, il exécute le même algorithme que dans A. Lorsqu'un processus effectue une opération de mise à jour, le résultat de la fonction `prepare` est transmis à la fonction `effect` en un seul message transmis sur le réseau.

Grâce à cette transformation, nous pouvons voir le modèle opérationnel comme une restriction du modèle *wait-free*, où des contraintes supplémentaires sur l'utilisation des messages ont été ajoutées. Par un léger abus de langage, nous utiliserons le terme « algorithme » (sans précision), pour faire référence aux algorithmes du modèle *wait-free* dans le chapitre 6. D'autres notions définies sur les algorithmes sont étendues aux algorithmes du modèle opérationnel grâce à l'injection canonique.

```
1 objet op2wf(A)
2     query q
3         return q(payload)
4     update u
5         data ← prepare_u(payload);
6         broadcast M(u, data);
7     Quand un message M(u, data) est reçu depuis p_i
8         payload ← effect_u(data, payload);
```

Algorithme 1 : Traduction d'un algorithme du modèle opérationnel dans le modèle *wait-free*

Chapitre 4

Le modèle *multi-thread* et la question de l'universalité du consensus [BMP19b]

Afin d'apporter un premier élément de réponse quand à la faisabilité des constructions universelles en système ouvert, il convient d'étudier le cas de l'objet consensus [BT85], qui fut à la base des constructions universelles. De plus, étant donné que c'est un objet avec un numéro de consensus infini, il est au sommet de la hiérarchie de Herlihy. L'étudier dans le cadre du modèle M_3 permet donc de vérifier si, oui ou non, il est toujours au sommet des objets en terme de puissance de coordination.

Énoncé du problème Dans les systèmes multi-threads, de nouveaux processus peuvent être créés et démarrés au cours de l'exécution, donc bien que le nombre de processus à chaque instant soit fini, il n'y a pas de limite sur le nombre total de processus pouvant participer à des exécutions de longue durée.

Le but de ce chapitre est d'étendre l'universalité du consensus au modèle d'arrivées infinies. Nous savons que des solutions au problème du consensus ont déjà été étudiées pour le modèle d'arrivées infinies. La question est donc « est-il possible de bâtir une construction universelle linéarisable *wait-free* basée sur des objets de consensus et des registres atomiques en lecture / écriture ? » Ce n'est pas trivial pour différentes raisons. Premièrement, bien que les constructions universelles *lock-free* fonctionnent toujours dans le modèle d'arrivées infinies parce qu'elles garantissent une condition de progression globale, ce n'est plus le cas pour les constructions universelles *wait-free* (en effet si une construction universelle permet que des processus soient bloqués indéfiniment, tant que certains avancent, il suffit que tous les nouveaux processus arrivant soient bloqués pour

que la condition soit respectée). Deuxièmement, les implémentations *wait-free* reposent sur ce qu'on appelle le mécanisme d'entraide. L'une des difficultés du modèle d'arrivées infinies est que le mécanisme d'entraide n'est pas évident à mettre en place. En effet, aider nécessite au moins qu'un processus devant être aidé soit capable d'annoncer son existence à d'autres processus désireux de l'aider. En raison du nombre infini de participants potentiels au fil de l'exécution, il n'est pas raisonnable de supposer que chaque processus peut écrire dans un registre dédié, et d'exiger des processus voulant aider de les lire tous. Lorsque seuls les objets consensus et les registres de lecture / écriture sont accessibles à un processus, un processus nouvellement arrivé est en concurrence avec un nombre potentiellement infini d'autres processus sur un même objet de consensus ou sur un même emplacement de mémoire, et peut échouer à toutes ses tentatives.

Contributions Ce chapitre a deux contributions principales (présentées à la conférence DISC2019 [BMP19b]).

— De la même manière que [FK14] qui propose d'abord un objet *Collect* qui sera utilisé comme bloc de construction pour une construction universelle, nous proposons d'abord une construction qui implémente un journal faiblement cohérent. Ce journal est utilisé comme une liste de présence où un processus qui arrive s'inscrit. Nous proposons deux implémentations d'un journal faiblement cohérent utilisant respectivement des objets consensus et les instructions matérielles spécifiques compare&swap.

— Nous proposons une construction universelle basée sur les objets consensus et l'objet journal faiblement cohérent dans le modèle d'arrivées infinies. Cela prouve que le consensus est universel même dans ce modèle.

Organisation Le reste de ce chapitre est organisé comme suit. Nous présentons d'abord une nouvelle abstraction, appelée le journal faiblement cohérent dans la section 4.1.
Dans la section 4.2, nous montrons comment le journal faiblement cohérent peut être utilisé conjointement avec le consensus pour mettre en œuvre une construction universelle *wait-free*.
Dans la section 4.3, nous donnons une implémentation du journal faiblement cohérent utilisant des objets de type consensus.

Dans la section 4.4, nous examinons un algorithme plus simple qui utilise compare&swap au lieu du consensus pour implémenter un journal faiblement cohérent.
Enfin, la section 4.5 conclut le chapitre.

4.1 L'abstraction du journal faiblement cohérent

Comme expliqué précédemment, la principale difficulté pour créer une construction universelle *wait-free* dans le modèle d'arrivées infinies réside dans le remplacement des mécanismes d'entraide déjà en place dans les algorithmes pour les modèles d'arrivées plus restrictifs. Dans cette section, nous présentons le journal faiblement cohérent, une abstraction qui permet à chaque processus d'annoncer sa propre valeur aux processus déjà présents.

Dans le modèle d'arrivées infinies, les objets linéarisables persistants (en opposition aux tâches) peuvent être exprimés sous forme de tâches distribuées, car un processus qui appelle plusieurs opérations sur un objet persistant peut être modélisé comme plusieurs processus émettant chacun une opération unique [CRR17]. Nous définissons maintenant l'abstraction du journal faiblement cohérent comme une tâche distribuée.

Dans une instance du journal faiblement cohérent, chaque processus p_i propose une valeur via une opération `append`(v_i), qui retourne la séquence de toutes les valeurs précédemment ajoutées. Le journal faiblement cohérent est *wait-free* mais pas linéarisable, en ce sens qu'il pourrait ne pas y avoir d'inclusion entre les séquences renvoyées par différents processus. Au lieu de cela, il est spécifié que la valeur proposée par un processus correct apparaîtra à terme dans toutes les séquences suivantes. De plus, l'ordre des valeurs à l'intérieur des séquences est cohérent sur les différentes séquences renvoyées par tous les processus.

Définition 19 (journal faiblement cohérent)**.** *Tous les processus $p_0, p_1, \ldots$ proposent des valeurs distinctes $v_0, v_1, \ldots$ en appelant* `append`(v_i)*, qui retourne une séquence finie $w_i = w_{i,1} \cdot w_{i,2} \cdots w_{i,|w_i|}$ telle que :*

Validité *Toutes les valeurs d'une séquence w_i ont été ajoutées par un processus : $\forall j, \exists k : w_{i,j} = v_k$.*

Suffixe *Si le processus p_i termine son invocation, alors sa valeur est ajoutée à la fin de la séquence retournée : $w_{i,|w_i|} = v_i$*

Ordre total *Si deux processus p_i et p_j terminent leur invocation, alors toutes les paires de valeurs que w_i et w_j contiennent toutes les deux apparaissent dans le même ordre : il n'y a pas k_i, k_j, l_i, l_j tels que $w_{i,k_i} = w_{j,k_j}$, $w_{i,l_i} = w_{j,l_j}$, $k_i \leq l_i$ et $k_j > l_j$.*

Visibilité à terme *Si un processus p_i termine son invocation, alors, au bout d'un moment, tous les processus terminant leur invocation renverront une séquence contenant v_i. En d'autres termes, le nombre de séquences retournées qui ne contiennent pas v_i est fini.*

Wait-freedom *Aucun processus ne prend un nombre infini d'étapes dans une exécution.*

4.2 Une construction universelle

L'algorithme 2 présente une construction universelle utilisant un journal faiblement cohérent et des objets de type consensus. Cet algorithme est similaire à celui présenté dans [HS08], à l'exception du tableau de registres à écrivain unique/lecteurs multiples utilisés par les processus pour annoncer leurs opérations qui est remplacé par un journal faiblement cohérent. L'objet partagé à implémenter est représenté par un état initial initialState et un ensemble d'opérations qui modifient l'état de l'objet et renvoient une valeur.

Les processus partagent deux variables :

— announce est un journal faiblement cohérent dans lequel les processus ajoutent leur propre invocation ;

— operations est un objet consensus à la fin d'une liste chaînée d'opérations. La liste est une succession de nœuds de la forme $\langle v, \mathsf{cons} \rangle$, où v est l'invocation d'un processus et cons est un objet de type consensus, référençant un autre nœud après que le consensus ait été gagné par un processus.

Lorsque le processus p_i appelle apply(invoc), il ajoute d'abord invoc à announce et obtient une liste toHelp_i d'invocations en retour. Ensuite, il tente d'insérer les invocations de toHelp_i à la fin de la liste operations jusqu'à ce que toutes les invocations de announce aient été insérées. En parcourant la liste, il conserve

un état state_i de l'objet implémenté, initialisé à initialState et sur lequel toutes les invocations sont appliquées dans leur ordre d'apparition dans la liste.

```
   operation apply(invoc) is
1      toHelp_i ← announce.append(invoc);
2      cons_i ← operations;
3      state_i ← initialState;
4      while toHelp_i ≠ ε do
5          ⟨winner_i, cons_i⟩ ← cons_i.propose(⟨toHelp_i[0], ⊥⟩);
6          toHelp_i ← toHelp_i \ winner_i;
7          if winner_i = invoc then result_i ← state_i.invoke(winner_i);
8          else state_i.invoke(winner_i);
9      return result_i;
```

Algorithme 2 : Construction universelle *wait-free* dans le modèle d'arrivées infinies

Nous prouvons maintenant que l'algorithme 2 est linéarisable et *wait-free*. La linéarisation est obtenue par l'algorithme 2 de la même manière que dans [HS08], donc la preuve du Lemme 1 est similaire.

Lemme 1 (Linéarisation)**.** *Toutes les exécutions admissibles par l'algorithme 2 sont linéarisables.*

Démonstration. Soit α une exécution admissible par l'algorithme 2. Remarquons d'abord que, pour toute opération apply(invoc) invoquée par le processus p_i, au plus un emplacement mémoire cons est tel que cons.head = invoc. En effet, supposons que ce ne soit pas le cas, et considérons les deux premiers emplacements de mémoire, cons_j et cons_k. Les deux ont été proposés en ligne 5 par les processus p_j et p_k. Comme les opérations sont totalement ordonnées dans une liste, le processus p_k accède à cons_j avant d'accéder à cons_k. Après avoir accédé à cons_j, $\mathsf{invoc} \notin \mathsf{toHelp}_k$ en conséquence de la ligne 6, ce qui contredit le fait que p_k proposait invoc sur cons_k.

Définissons le point de linéarisation de toute opération apply(invoc) pour laquelle il existe un emplacement mémoire cons tel que cons.head = invoc comme la première étape au cours de laquelle certains processus ont appelé cons.propose.

Nous prouvons maintenant que toute opération apply(invoc) effectuée par un processus qui termine p_i a un point de linéarisation, entre son point d'invocation et de fin. Par la propriété *validité* de announce, et comme toutes les valeurs invoc

sont différentes, aucun processus ne propose invoc avant que p_i n'arrive dans le système.

Par la propriété du *suffixe* de announce, au début de la boucle de p_i, invoc $\in$ toHelp$_i$. Lorsque p_i se termine, invoc $\notin$ toHelp$_i$. Par conséquent, invoc a été supprimée en ligne 6 d'une itération de la boucle, et cons au début de l'itération est tel que cons.head = invoc à la fin de la boucle.

Enfin, les opérations sont appliquées par p_i sur s dans le même ordre dans lequel elles apparaissent dans la liste (Lignes 7 et 8), qui est le même ordre que leurs points de linéarisation, ce qui conclut la preuve. □

La preuve de la propriété de *wait-freedom* (Lemme 2) est plus difficile car la preuve de [HS08] repose fortement sur le fait que le nombre de processus est fini.

Lemme 2 (*Wait-freedom*)**.** *Toutes les exécutions admissibles par l'algorithme 2 sont wait-free.*

Démonstration. Supposons qu'il existe une exécution α admissible par l'algorithme 2 qui n'est pas *wait-free*. Cela signifie que certains processus p_i exécutent un nombre infini d'étapes dans α. Par la propriété de *wait-freedom* de announce, p_i entre dans la boucle while après un nombre fini d'étapes, et chaque itération de la boucle se termine. Par conséquent, p_i exécute un nombre infini d'itérations de boucle. Soit $w_i = w_{i,1} \cdot w_{i,2} \cdots w_{i,|w_i|}$ la valeur initiale de toHelp$_i$. Comme w_i est fini et p_i propose $\langle w_{i,k}, \perp \rangle$ à chaque itération, il existe une valeur k telle que p_i propose $\langle w_{i,k}, \perp \rangle$ un nombre infini de fois.

Soit $win_0, win_1, \ldots$ la séquence infinie des valeurs prises par winner$_i$ lors de l'exécution, soit $p_{\omega(j)}$ le processus qui a exécuté le pas de calcul en ligne 5 écrivant la valeur win_j dans le consensus et que $p_{a(j)}$ soit le processus qui a appelé apply(win_j).

Comme les processus $w_{\omega(j)}$ proposent toujours la première invocation de toHelp$_{\omega(j)}$ qui n'a pas encore été insérée dans la liste, il y a un nombre infini de valeurs win_j telles que, soit

— $w_{i,k}$ ne fait pas partie de $w_{\omega(j)}$ ou

— $w_{i,k}$ fait partie de $w_{\omega(j)}$, mais apparaît après win_j dans la liste.

Par la propriété de *visibilité à terme*, le premier cas ne concerne qu'un nombre fini de win_j, il y a donc un nombre fini de valeurs win_j dans le second cas.

Pour chacun d'eux, par la propriété de *suffixe*, $win_j = w_{a(j),|w_{a(j)}|}$, c'est à dire que le processus qui a appelé `apply`(win_j) a obtenu win_j à la dernière invocation de sa liste $\mathsf{toHelp}_{a(j)}$. Par la propriété d'*ordre total*, il est impossible que $p_{\omega(j)}$ obtienne win_j avant $w_{i,k}$, et $p_{a(j)}$ obtient $w_{i,k}$ avant win_j. Par conséquent, $w_{a(j)}$ ne contient pas $w_{i,k}$. Cependant, cela contredit la propriété de *visibilité à terme* qui empêche un nombre infini de $p_{a(j)}$ processus d'ignorer $w_{i,k}$.

Il s'agit d'une contradiction, ce qui signifie que l'hypothèse d'une exécution qui n'est pas *wait-free* est absurde. □

4.3 Du consensus au journal faiblement cohérent

La principale difficulté dans la mise en œuvre d'un journal faiblement cohérent réside dans l'allocation d'un emplacement mémoire par processus, où il peut annoncer en toute sécurité son invocation d'opération. Comme il est impossible d'allouer un tableau infini en une fois, il est nécessaire de construire une structure de données dans laquelle les processus allouent leur propre mémoire et la rendent accessible à d'autres processus, en gagnant un consensus. La liste de l'algorithme 2 suit un modèle similaire, mais il pose un défi : comme un nombre infini de processus accèdent à la même séquence d'objets consensus, un processus peut perdre toutes ses tentatives pour insérer son propre nœud, brisant la propriété de *wait-freedom*.

L'algorithme 3 résout ce problème en utilisant une nouvelle fonctionnalité, que nous appelons l'*aide passive* : quand un processus gagne un consensus, il crée une liste latérale pour héberger les valeurs de processus concurrents sur le même objet de consensus. Comme seul un nombre fini de processus sont arrivés dans le système lorsque le consensus est obtenu, un nombre fini de processus tenteront d'insérer leur valeur dans la liste latérale, ce qui assure la terminaison de tout processus. La figure 4.1 présente une exécution de l'algorithme 3.

Les processus exécutant l'algorithme 3 partagent deux variables : first et last définies comme suit.

— first est un objet de type consensus qui accepte des valeurs de la forme $\mathsf{list} = \langle \mathsf{list.head}, \mathsf{list.tail} \rangle$, où list.tail est un objet consensus qui stocke des valeurs du même type que first.

```
   operation append(v) is
       // ajoute v au journal
1      node_i ← last.read().propose(⟨⟨v, ⊥⟩, ⊥⟩);
2      last.write(node_i.tail);
3      while node_i.head ≠ v do
4          node_i ← node_i.tail.propose(⟨v, ⊥⟩);
       // parcours le journal
5      log_i ← ε ; list_i ← first ; node_i ← list_i.head;
6      while true do
7          log_i ← log_i ⊕ node_i.head;
8          if node_i.head = v then return log_i;
9          node_i ← node_i.tail;
10         if node_i = ⊥ then
11             list_i ← list_i.tail;
12             node_i ← list_i.head ;
```

Algorithme 3 : Journal faiblement cohérent, *wait-free*, utilisant le consensus

list.head est un noeud de la liste latérale de la forme node = ⟨node.head, node.tail⟩, où node.head est une valeur ajoutée par un processus et node.tail est un objet de consensus contenant valeurs du même type que node. En d'autres termes, first référence le début d'une liste de listes de valeurs ajoutées.

— last est un registre de lecture/écriture référençant un objet de type consensus du même type que first, et initialisé à l'adresse de first. En l'absence de concurrence, last fait référence à la fin de la liste commençant par first.

Lorsque le processus p_i invoque append(v), il lit d'abord last, puis propose une liste contenant v comme son successeur et il écrit la valeur renvoyée par le consensus dans last. Si p_i perd le consensus, il insère v dans la liste latérale du successeur (lignes 3 et 4). Après cela, p_i parcourt la liste des listes pour construire la séquence log_i qu'elle renvoie ($\oplus$ représente la concaténation).

Notons que le consensus et l'écriture sur les lignes 1 et 2 ne se font pas atomiquement. Cela signifie qu'une très ancienne valeur peut être écrite dans last, auquel cas sa valeur pourrait reculer. La propriété centrale de l'algorithme, prouvée par le lemme 3, est que last finit à terme par avancer, permettant aux processus très lents de trouver une place dans une liste latérale et donc de terminer.

Lemme 3. *Si un nombre infini de processus exécutent la ligne 2, alors le nombre de processus qui lisent la même valeur du registre* last *à la ligne 1 est fini.*

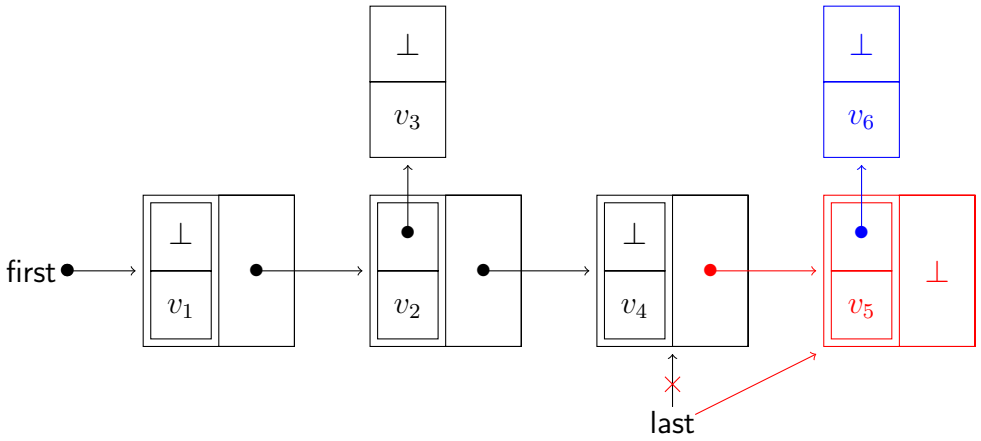

FIGURE 4.1 – Dans cet exemple, p_5 qui insère v_5 et p_6 qui insère v_6 lisent tous deux la même valeur $\langle\langle v_4, \bot\rangle, \bot\rangle$ de last. Le processus p_5 gagne le consensus et insère v_5 après v_4, et p_6 perd le consensus, donc il insère v_6 dans la liste latérale créée par p_5.

Démonstration. Prouvons d'abord par induction sur la liste des listes du journal faiblement cohérent que pour toutes valeurs list $= \langle$list.head, list.tail$\rangle$, le nombre d'opérations `write` de list dans last à la ligne 2 est fini.

- Initialement, first n'est jamais écrit dans last, car seules les valeurs décidées à la ligne 1 sont écrites, et first n'est jamais proposé.
- Prouvons maintenant que, si le nombre d'écritures de la valeur list dans last est fini, alors le nombre d'écritures de list.tail dans last est fini aussi.
  Nous prouvons la proposition contraposée suivante : si le nombre d'écritures de list.tail dans last est infini, alors le nombre d'écritures de list dans last est également infini.
  Pour écrire list.tail dans last, un processus doit lire list depuis last à la ligne 1. Comme list.tail est écrit un nombre infini de fois et list est lu un nombre infini de fois, alors nécessairement, list est écrit un nombre infini de fois aussi.

Supposons maintenant qu'un nombre infini de processus exécutent la ligne 2, et qu'un nombre infini de lectures de last retournent list. Cela implique qu'il y ait eu un nombre infini d'opérations `write` de list dans last à la ligne 2, ce qui contredit le résultat de l'induction précédente. □

Lemme 4 (Validité)**.** *Toutes les valeurs de la séquence w_i ont été ajoutées par un processus.*

Démonstration. La valeur log retournée par l'algorithme, est construite par concaténation de valeurs node.head, qui ne peuvent être créées qu'aux lignes 1 ou 4, en utilisant une valeur ajoutée préalablement. □

Lemme 5 (suffixe)**.** $w_{i,|w_i|} = v_i$

Démonstration. Ceci est une conséquence directe du fait que v_i soit ajouté à la fin de log_i à la ligne 7 juste avant l'instruction de retour sur la ligne 8. □

La définition 20 formalise l'ordre dans lequel les valeurs sont ordonnées dans le journal faiblement cohérent. Intuitivement, cet ordre est la concaténation de toutes les listes latérales. Dans l'algorithme 3, la liste de liste est parcourue dans cet ordre de priorité, ce qui garantit la cohérence de l'ordre de toutes les séquences retournées (propriété d'*Ordre Total* du journal faiblement cohérent).

Définition 20 (Préséance)**.** *Une valeur v_i précède une valeur v_j dans le journal faiblement cohérent si :*

- *v_i est dans une* list *l_i et v_j est dans une* list *l_j de telle sorte qu'il existe une séquence de listes $\{l_1, \ldots, l_n\}$ telle que pour toutes l_k, $l_k = \langle node_k, l_{k+1} \rangle$, et $l_i = \langle node_i, l_1 \rangle \wedge l_n = \langle node_n, l_j \rangle$.*
- *v_i et v_j sont dans la même* list *l et il existe une séquence de nœuds $\{n_1, , n_n\}$ telle que pour tous n_k, $n_k = \langle v, n_{k+1} \rangle$, $n_1 = \langle v_i, n_2 \rangle$ et $n_n = \langle v_j, n_{n+1} \rangle$.*

Nous étendons la notion de préséance aux listes et nœuds en utilisant la même définition.

Lemme 6 (Ordre Total)**.** *Si deux processus p_i et p_j terminent leurs invocations, alors toutes les paires de valeurs que w_i et w_j contiennent toutes les deux apparaissent dans le même ordre.*

Démonstration. Remarquons que les deux processus p_i et p_j ajoutent des valeurs dans leur journal suivant l'ordre de préséance défini par la définition 20. Par conséquent, pour deux valeurs quelconques v_k et v_l qui sont toutes deux contenues par w_i et w_j, p_i et p_j les ont ajoutés à la fin du journal dans le même ordre, ce qui prouve le lemme. □

Lemme 7 (Visibilité à terme)**.** *Si un processus p_i termine son invocation, le nombre de séquences retournées qui ne contiennent pas v_i est fini.*

Démonstration. Notons l_i et n_i les valeurs de list_i et node_i lorsque p_i se termine.

Supposons (par contradiction) qu'il existe un nombre infini de processus dont la séquence retournée ne contient pas v_i, et un nombre infini d'entre eux ont commencé leur opération après le retour de p_i. Pour chaque processus p_j, soient l_j et n_j les valeurs de list_j et node_j lorsque p_j termine son exécution. Comme la boucle de collecte respecte l'ordre de préséance de la définition 20, pour un nombre infini de p_j, l_j précède l_i. Comme il n'y a qu'un nombre fini de listes précédant l_i (p_i se termine), un nombre infini de processus ont la même valeur l de l_j. Tous lisent la même valeur de last à la ligne 1 et ont écrit à la ligne 2. Cela contredit le lemme 3. □

Lemme 8 (*Wait-freedom*)**.** *Aucun processus ne prend un nombre infini d'étapes dans une exécution.*

Démonstration. Supposons qu'il existe une exécution α telle que le processus p_i effectue un nombre infini d'étapes dans α en essayant d'ajouter v_i au journal faiblement cohérent. Cela signifie que l'une des deux boucles (lignes 3 et 6) boucle un nombre infini de fois :

- Si la boucle à la ligne 3 boucle un nombre infini de fois, cela signifie que $\mathsf{node.head} \neq v_i$ pour un nombre infini de nœuds. Cela implique qu'un nombre infini de valeurs sont ajoutées dans la même liste à la ligne 4, ce qui signifie qu'un nombre infini de processus lisent la même valeur à la ligne 1, et écrivent à la ligne 2, ce qui contredit le lemme 3.
- Si la boucle à la ligne 6 boucle un nombre infini de fois, cela signifie que p_i ne lit jamais v_i, et comme il existe un nombre fini de listes qui précède la liste l_i dans laquelle v_i a été ajoutée, l'une de ces listes contient un nombre infini de nœuds. Tous ces nœuds ont été créés par des processus lisant la même valeur de last à la ligne 1, ce qui contredit le lemme 3.

Les deux cas conduisent à une contradiction, donc l'algorithme est *wait-free*. □

4.4 Du compare&Swap au journal faiblement cohérent

Dans les environnements multithreads, le consensus est généralement remplacé par des opérations spéciales comme le compare&swap. Dans cette section, nous

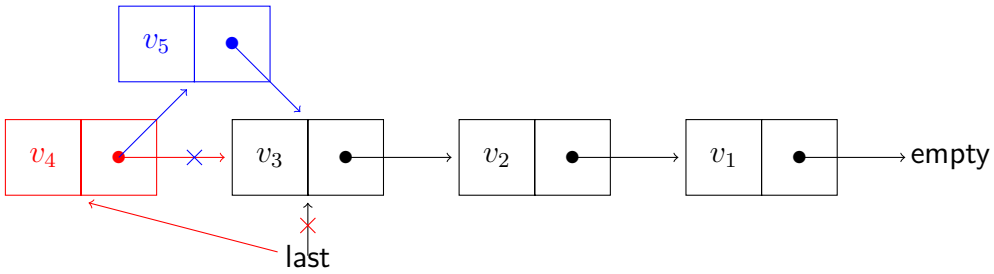

FIGURE 4.2 – Dans cet exemple, p_4 qui insère v_4 et p_5 qui insère v_5 lisent tous les deux la même valeur v_3 de last. Le processus p_4 gagne son compare&swap et insère v_4 avant v_3, et p_5 perd le compare&swap en faveur de p_4 et insère v_5 après v_4.

remplaçons les registres de lecture/écriture et les objets de type consensus par les registres CAS. D'une part, comme dans le modèle d'arrivées finies, les objets de type consensus et les registres CAS sont équivalents en termes de calcul : un objet de consensus peut être facilement simulé en implémentant $x.\texttt{propose}(v)$ en exécutant $x.\texttt{CAS}(\perp, v)$, et inversement, compare&swap peut être mis en œuvre en utilisant le consensus comme prouvé dans la section précédente. D'autre part, compare&swap est plus flexible car il permet d'écrire plusieurs fois dans le même registre.

L'algorithme 4 présente une implémentation simplifiée du journal faiblement cohérent basée sur le registre CAS au lieu d'objets consensus. Il existe deux différences principales entre l'algorithme 4 et l'algorithme 3. Tout d'abord, les valeurs sont stockées dans une liste au lieu d'une liste de listes. Deuxièmement, la liste est gérée comme une pile (FILO) et non comme une file (FIFO). La figure 4.2 illustre une exécution de l'algorithme.

Les processus partagent un registre CAS unique last, qui stocke soit la valeur initiale, empty, soit des nœuds de la forme $\langle \mathsf{head}, \mathsf{tail} \rangle$, où head est une valeur ajoutée par un processus et tail a le même type que last.

Lorsqu'un processus appelle $\texttt{append}(v)$, il essaie d'abord d'ajouter v en haut de la pile, en remportant un compare&swap sur last. Si cette tentative échoue et qu'une valeur v' a été insérée concurremment par p_j, p_i essaie continuellement d'insérer v après v'. De façon similaire à l'algorithme 3, p_i réussit finalement parce qu'il ne fait concurrence dans cette tâche qu'avec un ensemble fini de processus qui lisent last avant que p_j ne gagne son compare&swap. Après que p_i ait inséré

```
   operation append(v) is
       // ajoute v au journal
1      next_i ← last;
2      if ¬last.CAS(next_i, ⟨v, next_i⟩) then
3          repeat
4              first_i ← next_i;
5              next_i ← first_i.tail;
6          until first_i.tail.CAS(next_i, ⟨v, next_i⟩);
       // parcourt le journal
7      log_i ← v;
8      while next_i ≠ empty do
9          log_i ← next_i.head ⊕ log_i;
10         next_i ← next_i.tail;
11     return log_i;
```

Algorithme 4 : Journal faiblement cohérent, *wait-free*, utilisant compare&swap

v dans la liste, il parcourt la liste jusqu'à la fin pour construire la séquence de retour. La traversée est effectuée en dernier entré, premier sorti, donc les valeurs lues sont ajoutées au début de log_i.

Lemme 9 (Validité)**.** *Toutes les valeurs d'une séquence w_i ont été ajoutées par un processus.*

Démonstration. Toutes les valeurs d'une séquence w_i ont été ajoutées à la ligne 9 et v (valeur ajoutée par le processus) à la ligne 7.

v a est la valeur qui vient d'être ajoutée par p_i. toutes les autres valeurs sont des valeurs présentes dans le journal, ajoutées par des processus gagnant un compare&swap aux lignes 2 et 6. □

Lemme 10 (Suffixe)**.** *Si le processus p_i termine son invocation, alors sa valeur v_i est ajoutée à la fin de la séquence retournée.*

Démonstration. la valeur v_i ajoutée est la première à être ajoutée ligne 7, puis lors du parcours du journal, toute les valeurs lues sont ajoutées avant les valeurs précédentes (ligne 9). □

Définition 21 (Préséance)**.** *Une valeur v_i précède une valeur v_j dans le journal faiblement cohérent si il existe une séquence de noeuds $\{n_1, \ldots, n_n\}$ telle que pour tous n_k, $n_k = \langle v, n_{k+1}\rangle$, $n_1 = \langle v_j, n_2\rangle$ et $n_n = \langle v_i, n_{n+1}\rangle$.*

Lemme 11 (Odre total)**.** *Si deux processus p_i et p_j terminent leurs invocations, alors toutes les paires de valeurs que w_i et w_j contiennent toutes les deux apparaissent dans le même ordre.*

Démonstration. Remarquons que les deux processus p_i et p_j ajoutent des valeurs dans leur journal suivant l'ordre de préséance défini par la définition 21. Par conséquent, pour deux valeurs quelconques v_k et v_l qui sont toutes deux contenues par w_i et w_j, p_i et p_j les ont ajoutés au début du journal dans le même ordre. En effet, une fois une valeur ajoutée dans le journal faiblement cohérent, il est possible que des noeuds soient ajoutés avant ou après, mais l'ordre de préséance ne sera jamais modifié □

Lemme 12 (Visibilité à terme)**.** *Si un processus p_i termine son invocation, alors, à terme, tous les processus terminant leur invocation renverront une séquence contenant v_i*

Démonstration. La preuve est la même que celle du lemme 7 avec la définition modifiée de l'odre de préséance (définition 21). □

Lemme 13 (*Wait-freedom*)**.** *Aucun processus ne prend un nombre infini d'étapes dans une exécution.*

Démonstration. De manière similaire à l'algorithme 3, il ne peut y avoir qu'un nombre fini de processus essayant de gagner le compare&swap à la ligne 2. Une fois qu'un processus p_i gagne le compare&swap à cette ligne, tous les processus essayant d'ajouter une valeur liront la valeur de last gagnée par p_i, qui est différente. Il y a donc une boucle finie pour le reste des processus qui étaient en concurrence avec p_i. Donc tout processus termine. □

4.5 Conclusion

Le consensus est un problème central en informatique distribuée, car il permet des implémentations linéarisables *wait-free* de tous les objets avec une spécification séquentielle, dans des systèmes composés de n processus asynchrones pouvant tomber en panne. Dans ce chapitre, nous avons posé la question de savoir si le résultat tient toujours dans le modèle d'arrivées infinies, dans lequel un nombre potentiellement infini de processus peut arriver et partir lors d'une

exécution. Nous avons répondu positivement à cette question en donnant une construction universelle *wait-free* et linéarisable utilisant uniquement des objets de type consensus et des registres de lecture/écriture.

Théorème 1. *Le consensus multi-valué est universel dans le modèle M_3.*

Remarque 2. *Le registre compare&swap est également universel dans le modèle M_3.*

Notre construction proposée est basée sur deux listes contenant toutes les opérations appelées sur l'objet. Une question intéressante est de savoir s'il est possible de réduire le coût en mémoire en supprimant éventuellement les opérations des deux listes. Notre approche ne permet pas de telles optimisations, car elle repose sur un mécanisme « d'aide passive », dans lequel les processus remportant une instance de consensus allouent des emplacements de mémoire pour héberger la valeur des potentiels processus qui perdraient toutes les instances de consensus dans lesquelles ils sont impliqués.

Chapitre 5

Extension de la hiérachie des modèles *wait-free* aux modèles ouverts [PMB20]

Nous savons maintenant que le consensus est toujours au plus haut niveau en terme de calculabilité, mais qu'en est-il de tous les objets ayant un numéro de consensus infini dans la hiérarchie de Herlihy ?

Ce chapitre explore la puissance de synchronisation des objets partagés dans les systèmes multi-threads en étendant la fameuse hiérarchie *wait-free* des objets de Herlihy [Her91] pour tenir compte des contraintes apportées par les systèmes ouverts.

Une spécificité des systèmes multi-threads doit être prise en compte. Les threads partagent un espace mémoire commun dans lequel ils peuvent allouer un nombre (virtuellement) illimité mais fini d'emplacements de mémoire pour instancier des structures de données d'enregistrement ou des tableaux finis. En particulier, lorsqu'aucune limite n'est connue sur le nombre de threads dans une exécution, l'attribution d'un registre à chacun d'eux n'est pas anodine. Ce fait est souvent considéré comme secondaire lors de la conception d'algorithmes distribués. Par exemple, [Agu04] identifie comme « trivial » le remplacement de tableaux finis indexés par les processus en tableaux infinis ou listes chaînées.

Une conséquence de ce chapitre est que le maintien de structures de données extensibles telles que les listes chaînées nécessite une puissance de synchronisation qui n'est pas nécessairement fournie par tous les objets qui ont un numéro de consensus infini.

Les deux aspects mentionnés ci-dessus ont un impact important sur les algorithmes qui peuvent être mis en œuvre dans les systèmes multi-threads et ceux

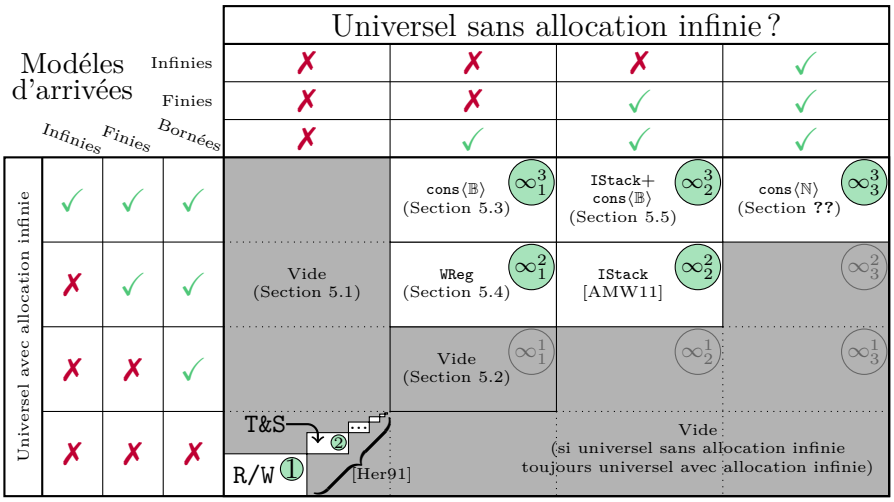

FIGURE 5.1 – Hiérarchie des modèles *wait-free* étendue : dans un système multi-thread, il est impossible d'implémenter un objet O_1 en utilisant un nombre quelconque d'instances de O_2 et des registres de lecture/écriture, si O_2 est plus à gauche, ou en bas, que O_1. Les cercles verts affichent le numéro de consensus de chaque degré

qui ne le peuvent pas, et donc sur le pouvoir de coordination des objets partagés : dans [AMW11], Afek, Morrison et Wertheim ont présenté un objet appelé *pile d'itérateurs* (noté `IStack`) qui a un numéro de consensus infini, mais ne peut pas être utilisé pour implémenter un consensus quand un nombre infini de processus peuvent rejoindre au fil du temps une exécution. Ce chapitre répond à la question suivante : comment comparer la puissance de synchronisation des objets partagés dans les systèmes multithreads ?

Approche. En suivant la même approche que dans [Her91], nous proposons de comparer la puissance de synchronisation des objets partagés en fonction du nombre maximal de processus qu'ils sont capables de synchroniser, y compris dans des situations où l'ensemble des processus participants est initialement inconnu ou peut changer lors d'une exécution. Les objets sont donc comparés selon leur puissance de synchronisation dans les quatres modèles d'arrivées définis dans le chapitre 3.

— Le modèle classique, M_1^n.
— Le modèle d'arrivées bornées, M_1.
— Le modèle d'arrivées finies, M_2.

— Le modèle d'arrivées infinies, M_3.

De plus, l'impossibilité d'allouer des tableaux infinis est un facteur limitant important qui restreint la puissance de calcul de certains objets dans les systèmes multithreads. Nous étudions également la puissance de synchronisation des objets partagés selon la possibilité ou non d'allouer des tableaux infinis. Par conséquent, nous proposons la hiérarchie bidimensionnelle présentée sur la figure 5.1.

Dans cette hiérarchie, les objets partagés sont triés horizontalement en fonction de leur universalité dans les modèles M_1^n, M_1, M_2 et M_3 lorsque l'allocation de mémoire infinie n'est pas disponible, et verticalement sur leur capacité à le faire quand il est possible d'allouer des tableaux infinis. Nous remettons ensuite en question la signification de cette hiérarchie en explorant s'il existe ou non un objet remplissant chaque degré possible.

Contributions : Dans un premier temps, nous montrons comment la hiérarchie proposée englobe l'existant puis pour chaque nouveau degré, soit un objet représentatif est proposé, soit il se révèle vide.

Étendre la hiérarchie des modèles *wait-free*. Nous montrons que, d'une part, la hiérarchie proposée coïncide avec la hiérarchie de Herlihy sur des objets avec un nombre de consensus fini. En effet, le théorème 2 prouve que les tableaux infinis ne sont pas nécessaires pour les constructions universelles dans les modèles M_1^n et M_1, ce qui justifie que nous gardions le même terme « numéro de consensus » pour catégoriser les objets partagés dans notre hiérarchie. En revanche, la hiérarchie proposée affine celle proposée par Herlihy pour les objets à numéro de consensus infini. On dit qu'un objet O a le numéro de consensus ∞_x^y, pour $x, y \in \{1, 2, 3\}$ si O est universel dans M_x mais pas M_{x+1} (si $x \neq 3$) lorsque l'allocation de mémoire infinie n'est pas disponible, et O est universel dans M_y mais pas M_{y+1} (si $y \neq 3$) lorsqu'une allocation de mémoire infinie est disponible. Étant donné que l'accès à des tableaux infinis n'est jamais préjudiciable, aucun objet n'a le numéro de consensus ∞_x^y pour $y < x$.

Remarquons que la hiérarchie proposée n'est pas stricte : il est impossible d'utiliser un nombre quelconque d'objets de numéro de consensus ∞_1^3 pour implémenter un objet avec le numéro de consensus ∞_2^2 dans un système multi-thread, car cela nécessiterait l'allocation de tableaux infinis. Inversement, il est impossible d'implémenter un objet avec le numéro de consensus ∞_1^3 en utilisant uniquement

des objets avec le numéro de consensus ∞_2^2 car cela pourrait entrainer une situation de famine pour certains processus participants tandis que de nouveaux processus arrivent constamment dans le système.

Identifier tous les degrés remplis. En suivant notre approche, nous prouvons qu'aucun objet n'a le numéro de consensus ∞_1^1 (Théorème 3), et nous identifions des objets remplissant tous les degrés restants de la hiérarchie, comme le montre la figure 5.1. Nous savons que le consensus multi-valué (noté $\texttt{cons}\langle\mathbb{N}\rangle$) est toujours universel dans tous les modèles considérés dans ce chapitre, c'est-à-dire qu'il a le numéro de consensus ∞_3^3 d'après le résultat du chapitre 4 (c'est aussi le cas de l'objet compare&swap). En reformulant les théorèmes concernant la pile d'itérateurs [AMW11], nous en déduisons naturellement que les piles d'itérateurs ont le numéro de consensus ∞_2^2. De façon intéressante, nous prouvons que le consensus binaire (noté $\texttt{cons}\langle\mathbb{B}\rangle$) n'est pas universel dans les systèmes multi-threads, résultant en un numéro de consensus de ∞_1^3 (Théorème 4). La preuve que la composition du consensus binaire et des piles d'itérateurs a le numéro de consensus ∞_2^3 (Théorème 6) est la partie la plus technique du chapitre. Nous montrons qu'un registre fenêtré (noté `WReg`), dans lequel une opération de lecture renvoie les k dernières valeurs écrites (k choisies à l'initialisation), a le numéro de consensus ∞_1^2 (Théorème 5).

Organisation. Le reste de ce chapitre est organisé comme suit. Les sections 5.1 et 5.2 identifient les degrés vides de la hiérarchie en prouvant les théorèmes 2 et 3. Les sections 5.3, 5.4 et 5.5 montrent que les degrés restants ne sont pas vides en fournissant le numéro de consensus du consensus binaire, des registres fenêtrés et une composition du consensus binaire et des piles d'itérateurs. Enfin, la section 5.6 conclut le chapitre.

5.1 L'allocation infinie de mémoire n'est pas nécessaire dans M_1

L'article original sur la hiérarchie des modèles *wait-free* [Her91] ne mentionne aucune limitation qui pourrait survenir dans les modèles de calcul où l'allocation infinie n'est pas disponible. Dans cette section, nous prouvons que, dans le contexte des modèles d'arrivées bornées, l'allocation de mémoire infinie n'est pas

un facteur décisif pour déterminer si l'universalité peut être réalisée ou non. Cela implique que notre hiérarchie coïncide avec celle d'Herlihy pour les objets avec un numéro de consensus fini, ce qui justifie notre choix de garder le même nom.

Ce résultat s'appuie sur l'observation que, dans MA_1^n, tout algorithme *wait-free* du consensus binaire a une borne sur le nombre d'emplacements de mémoire utilisés par n'importe quelle exécution, tant qu'il existe une borne sur les identificateurs de processus (Lemme 14). Une telle limite peut être obtenue en utilisant des algorithmes de renommage. Par exemple, [Att+90] ne nécessite pas non plus d'allocation de mémoire infinie. Dans cette section, nous supposons, sans perte de généralité, qu'il existe une borne N sur les identificateurs de processus.

Lemme 14. *Pour tout objet O, si $\texttt{cons}\langle\mathbb{B}\rangle$ peut être implémenté dans $MA_1^n[O]$, alors $\texttt{cons}\langle\mathbb{B}\rangle$ peut être implémenté dans $M_1^n[O]$.*

Démonstration. Supposons qu'il existe un algorithme A qui implémente le consensus binaire dans $MA_1^n[O]$. Une entrée de A est composée d'un ensemble Π de (au plus) n processus pris dans $\{p_0, \ldots, p_N\}$, et d'une relation qui associe une entrée booléenne à chaque processus dans Π. Nous prouvons la proposition suivante :

Déclaration 1. *Pour chaque entrée possible $\langle \Pi \subset \{p_0, \ldots, p_N\}, \Pi \to \mathbb{B}\rangle$, un nombre fini de configurations peut être accédé quelle que soit l'exécution de A.*

Démonstration. Supposons que ce ne soit pas le cas. Nous construisons, récursivement, une exécution infinie dans laquelle un nombre infini de configurations est accessible à chaque étape. Initialement, soit $\alpha_0 = \varepsilon$, l'exécution ne contenant aucune étape. Supposons que nous ayons construit une exécution α_k contenant k étapes, de telle sorte qu'un nombre infini de configurations soient accessibles à partir de $C(\alpha_k)$. A partir de $C(\alpha_k)$, une étape correspond à l'étape suivante de l'un des processus qui n'a pas encore décidé, donc il y a au plus n différentes étapes possibles. Au moins l'un d'eux, β_k, conduit à une configuration à partir de laquelle un nombre infini de configurations est accessible. Posons $\alpha_{k+1} = \alpha_k\beta_k$.

Comme $(\alpha_k)_{k\in\mathbb{N}}$ est construit de telle sorte que α_k est un préfixe de α_{k+1}, il converge vers une exécution infinie $\alpha = \beta_0\beta_1\beta_2\ldots$ Certains processus exécutent un nombre infini d'étapes dans α, donc A n'est pas *wait-free*. C'est une contradiction. □

Le nombre d'entrées possibles est limité par $2^N \times 2^n$. Pour chacune d'elles, le nombre de configurations accessibles est fini par la proposition 1. Par conséquent, un nombre fini X_n de configurations est accessible par toute exécution de A. Dans chaque configuration, chaque processus peut être sur le point d'appeler une opération sur un objet partagé différent, donc au plus un nombre fini nX_n d'objets peut être utilisé par A. Par conséquent, A peut être simulé par un algorithme dans $M_1^n[O]$ qui alloue uniquement $n \times X_n$ emplacements de mémoire lors de la configuration initiale. □

Théorème 2. *Pour tout objet O, si $MA_1^n[O]$ est universel, alors $M_1^n[O]$ est également universel.*

Démonstration. Supposons que $MA_1^n[O]$ est universel ; par définition, $\texttt{cons}\langle\mathbb{B}\rangle$ peut être implémenté dans $MA_1^n[O]$. De par le Lemme 14, $\texttt{cons}\langle\mathbb{B}\rangle$ peut être implémenté dans $M_1^n[O]$. Il est possible d'implémenter $\texttt{cons}\langle\mathbb{N}\rangle$ en utilisant un nombre limité d'objets $\texttt{cons}\langle\mathbb{B}\rangle$ dans le modèle d'arrivées bornées en utilisant un algorithme comme celui donné dans [ZC09] et qui peut être facilement adapté à la mémoire partagée [Ray12]. Enfin, par le théorème 1, O est universel dans $M_1^n[O]$. □

Corollaire 1. *Pour tout objet O, si $MA_1[O]$ est universel, alors $M_1[O]$ est également universel.*

Démonstration. Supposons que $MA_1[O]$ est universel. Dans $M_1[O]$, nous pouvons utiliser la borne n donnée sur le nombre de processus et la construction universelle disponible dans $M_1^n[O]$ pour implémenter tout objet linéarisable *wait-free* dans $M_1[O]$. □

5.2 Aucun object ne possède le numéro de consensus ∞_1^1

Dans cette section, nous prouvons qu'aucun objet n'a le numéro de consensus ∞_1^1. Nous le prouvons en montrant que, lorsque l'allocation de mémoire infinie est disponible, tout objet O universel dans le modèle d'arrivées bornées est également universel dans le modèle d'arrivées finies. En effet, si $MA_1[O]$ est universel, il est possible d'utiliser des objets O pour résoudre le consensus parmi les n processus,

pour tout n. L'algorithme 5 utilise alors ces objets $\mathsf{cons}^n\langle\mathbb{N}\rangle$ pour résoudre le consensus dans MA_2 (Lemmes 15, 16 et 17).

Les processus partagent trois tableaux infinis greaterId, cons et adopt : pour chaque indice $r \in \mathbb{N}$, greaterId$[r]$ est un registre booléen, initialisé à **false**, qui ne peut être écrit par p_i que si $i \geq r$; cons$[r]$ est un objet $\mathsf{cons}^n\langle\mathbb{N}\rangle$ qui accepte la participation des processus $p_0, \ldots, p_{r-1}$; et adopt$[r]$ est un registre, initialisé à $\perp$, qui stockera la valeur décidée de cons$[r]$ afin que les processus $p_r, p_{r+1}, \ldots$ puissent savoir la valeur décidée sans participer.

L'algorithme 5 est basé sur des rondes. À la ronde r, les processus avec des identifiants inférieurs à r s'accordent sur une certaine valeur en utilisant l'objet $\mathsf{cons}^n\langle\mathbb{N}\rangle$ cons$[r]$, tandis que les autres processus annoncent simplement leur présence en marquant greaterId$[r]$. Si les premiers décident en premier, ils renvoient la valeur qu'ils ont décidée. Autrement, si ces derniers arrivent avant que le consensus n'ait lieu, d'autres tours sont nécessaires. Si les deux groupes écrivent simultanément, il est possible que certains processus décident d'une valeur à la ronde r tandis que d'autres commencent la ronde $r+1$. Dans ce cas, le protocole garantit qu'ils adoptent la valeur décidée pour les prochains cycles, garantissant ainsi un accord.

```
   operation propose(val_i) is
1  |  v_i ← val_i;
2  |  for r_i = 0, 1, 2, ... do
3  |  |  if i ≥ r_i then greaterId[r_i] ← true ;
4  |  |  else adopt[r_i] ← cons[r_i].propose(v_i) ;
5  |  |  if ¬greaterId[r_i] then return adopt[r_i] ;
6  |  |  if adopt[r_i] ≠ ⊥ then v_i ← adopt[r_i] ;
```

Algorithme 5 : Consensus dans le modèle $MA_2[\mathtt{cons}^n\langle\mathbb{N}\rangle]$ (code pour p_i)

Déclaration 2. *Pour toute ronde r, au plus r processus invoquent* `propose` *sur* cons$[r]$ *à la ligne 4.*

Démonstration. De par la ligne 3, un processus p_i ne peut exécuter la ligne 4 que si $i < r$, et s'il y a au plus r identifiants plus petits que r. □

Lemme 15 (*Wait-freedom*)**.** *Toutes les exécutions de l'algorithme 5 terminent dans MA_2.*

Démonstration. Dans MA_2, chaque exécution a un processus avec le plus grand identifiant (appelons-le i_{max}). La variable $\mathsf{greaterId}[r]$ n'est passée à **true** que si $r \leq i_{max}$ (Ligne 2), donc tous les processus se terminent au plus tard à la ronde $i_{max} + 1$ (ligne 6). □

Lemme 16 (Accord)**.** *Si les processus p_i et p_j décident respectivement v_i et v_j, alors $v_i = v_j$.*

Démonstration. Soit p_i un processus qui termine, en décidant v_i, avec le plus petit numéro de ronde r_i.

Nous démontrons d'abord, par récurrence sur r, que, pour tout $r > r_i$, tous les p_j participant à la ronde r commencent la ronde avec $v_j = v_i$. Supposons que p_j participe à la ronde $r = r_i + 1$. Pendant la ronde r_i, à la ligne 5, p_j lit $\mathsf{greaterId}[r_i] = \mathbf{true}$ et p_i lit $\mathsf{greaterId}[r_i] = \mathbf{false}$. Comme seul **true** est écrit dans $\mathsf{greaterId}[r_i]$ (ligne 3), p_i a exécuté la ligne 5 avant p_j, donc p_i a exécuté la ligne 4 avant que p_j n'exécute la ligne 6. Par conséquent, p_j a commencé la ronde $r_i + 1$ avec $v_j = v_i$. Supposons que la proposition soit valable pour $r > r_i$. De par la proposition 2 et la propriété de validité de $\mathsf{cons}^{r_i}\langle\mathbb{N}\rangle$, seul v_i peut être écrit dans $\mathsf{adopt}[r]$ à la ligne 4, donc, de par la ligne 6, tous les processus conservent leur valeur $v_j = v_i$, ou adoptent $\mathsf{adopt}[r] = v_i$, pour commencer la ronde numéro $r+1$.

Soit p_j un processus qui décide de v_j à la ronde $r_j \geq r_i$. Si $r_i = r_j$, comme $\mathsf{greaterId}[r_i]$ est **false** à la ligne 5 pour les deux, p_i et p_j invoquent $\mathsf{cons}[r_i].\texttt{propose}$ (Ligne 4) qui renvoie respectivement v_i et v_j. Par la proposition 2 et la propriété d'accord du r_i-consensus, $v_j = v_i$. Sinon, $r_j > r_i$. Tous les processus qui ont appelé $\mathsf{cons}[r_i].\texttt{propose}$ proposent v_i. Par la proposition 2 et la validité pour $\mathsf{cons}^{r_i}\langle\mathbb{N}\rangle$, $v_j = v_i$. □

Lemme 17 (Validité)**.** *Si p_i décide v, alors un processus a proposé v.*

Démonstration. Soit p_i un processus qui décide de v_i à la ronde r_i. Comme $\mathsf{greaterId}[r_i]$ est **false** à la ligne 5, $i < r_i$, donc p_i a écrit dans $\mathsf{adopt}[r_i]$ à la ligne 4. Supposons que v_i ne soit pas la valeur d'entrée d'un processus, et considérons la première fois qu'une valeur qui n'est pas l'entrée d'un processus est écrite dans v_j ou $\mathsf{adopt}[r_j]$ par un processus p_j. Cela ne peut pas se produire à la ligne 1 par définition de val_j. Par la proposition 2 et la validité pour $\mathsf{cons}^{r_j}\langle\mathbb{N}\rangle$, seule une valeur précédemment écrite dans v_j peut être écrite dans $\mathsf{adopt}[r_j]$ à la

ligne 4. Grâce à la condition, seule une valeur précédemment écrite dans $\mathsf{adopt}[r_j]$ peut être écrit dans v_j à la ligne 6. Enfin, seules les valeurs proposées par les processus peuvent être écrites dans $\mathsf{adopt}[r_i]$, y compris la valeur v_i retournée par p_i. □

Théorème 3. *Aucun objet n'a le numéro de consensus ∞_1^1.*

Démonstration. Supposons, par contradiction, qu'un objet O ait le numéro de consensus ∞_1^1. En particulier, $MA_1[O]$ est universel. Pour tout n, il est possible d'implémenter un objet $\mathsf{cons}^n\langle\mathbb{N}\rangle$ dans MA_2, en utilisant O par simulation de l'algorithme du modèle d'arrivées bornées, en définissant la borne à n.

Par les lemmes 15, 16 et 17, l'algorithme 5 est une implémentation du consensus dans MA_2, qui est universelle par le théorème 1. Par conséquent, le numéro de consensus de O est au moins ∞_1^2, ce qui est une contradiction. □

5.3 Le consensus booléen a le numéro de consensus ∞_1^3

Un objet a le numéro de consensus ∞_1^3, si une allocation de mémoire infinie est nécessaire pour le rendre universel dans le modèle d'arrivées finies, et suffisant dans le modèle d'arrivées infinies. L'une des raisons pour lesquelles cela pourrait se produire est que le nombre d'instances nécessaires à la synchronisation croît sans limite avec le nombre de processus. Récemment, [Ell+16] a introduit une hiérarchie basée sur la complexité classant les objets partagés en fonction du nombre d'instances nécessaires pour résoudre un consensus *obstruction-free*. Par exemple, au moins $\lceil \frac{n-1}{k} \rceil$ registres fenêtrés de taille k sont nécessaires pour résoudre le consensus multi-valué. De même, à notre connaissance, aucun algorithme connu utilise moins de $\log_2(n)$ objets consensus binaires pour résoudre le consensus [ZC09] dans le pire des cas. Afin d'être universel dans M_2, tout objet n'a que deux façons de contourner la limitation selon laquelle seul un nombre fixe et fini d'objets peut être créé lors de l'initialisation d'un algorithme : soit il a une complexité constante dans la hiérarchie d'Ellen et al, ou alors il fournit suffisamment de puissance de synchronisation pour entretenir une structure de données extensible (par exemple une liste chaînée), où de nouvelles instances de lui-même peuvent être créées au moment de l'exécution et être accessibles par les processus

nouvellement arrivés. Dans cette section, nous prouvons que $\texttt{cons}\langle\mathbb{B}\rangle$ est universel dans MA_3 (Proposition 1) mais pas dans M_2 (Proposition 2), c'est-à-dire que le consensus binaire a le numéro de consensus ∞_1^3 (Théorème 4).

Il a été démontré que l'objet sticky bit, une version réinitialisable du consensus binaire, est universel dans MA_1 dans [Plo89]. Des réductions du consensus à valeurs multiples au consensus binaire ont plus tard été proposées pour les systèmes à passage de messages [MRT00], et étendues à M_1 [Ray12]. L'algorithme 6 étend ce résultat au modèle MA_3. Les processus partagent trois tableaux infinis propose, isSet et cons : pour chaque index $j \in \mathbb{N}$, propose$[j]$ est destiné à stocker la valeur proposée par p_j, isSet$[j]$ est un booléen défini sur true uniquement après que propose$[j]$ ait été défini, et cons$[j]$ est un objet de consensus binaire dans lequel **true** est décidé si, et seulement si, la valeur de p_j est décidée. Lorsqu'un processus p_i propose une valeur val_i, il l'écrit d'abord dans proposed$[i]$ et définit isSet$[i]$ sur **true** pour annoncer sa valeur. Ensuite, il parcourt le tableau proposed dans l'ordre croissant des identifiants, jusqu'à ce qu'il convienne avec les autres participants d'une valeur proposed$[j]$ qui puisse être décidée.

```
   operation propose(val_i) is
1  |  proposed[i] ← val_i ; isSet[i] ← true;
2  |  for j = 0, 1, 2, ... do
3  |  |  if cons[j].propose(isSet[j]) then return proposed[j];
```

Algorithme 6 : Consensus dans $MA_3[\texttt{cons}\langle\mathbb{B}\rangle]$ (code pour p_i)

Proposition 1. *$MA_3[\texttt{cons}\langle\mathbb{B}\rangle]$ est universel.*

Démonstration. Nous prouvons que l'algorithme 6 implémente $\texttt{cons}\langle\mathbb{N}\rangle$ dans $MA_3[\texttt{cons}\langle\mathbb{B}\rangle]$.

Terminaison. Comme tous les processus écrivent **true** dans isSet$[i]$ (ligne 1) avant de lire isSet$[j]$ (ligne 3), le premier accès à isSet est une écriture par un processus p_{j_0}. Tous les processus p_i exécutant la boucle pour $j = j_0$ proposeront **true**, donc par la propriété de validité du consensus binaire, aucun processus n'exécute une itération pour $j > j_0$.

Accord. Par la propriété d'accord du consensus binaire, tous les processus décidant décident à la même ronde j, qui est le plus petit x tel que **true** soit décidé par cons$[x]$.

Validité. Supposons que p_i décide à la ronde j. Un processus lit $\mathsf{isSet}[j] =$ **true** sur la ligne 3, donc le processus p_j a précédemment écrit **true** à la ligne 1, après avoir écrit sa valeur proposée dans $\mathsf{proposed}[r_j]$ à la ligne 1. Il s'agit de la valeur renvoyée par p_i.

Enfin, par le théorème 1, $MA_3[\mathtt{cons}\langle\mathbb{N}\rangle]$ est universel. □

La proposition 2 ci-dessous montre que ni le consensus binaire ni les registres fenêtrés ne sont universels dans le modèle d'arrivées finies quand l'allocation de mémoire infinie est impossible. La preuve a le même fondement que les preuves dans [Ell+16], mais simplifiée car nous ne nous intéressons qu'à la décidabilité alors que leurs limites doivent être strictes. Plus précisément, la preuve de la proposition 2 construit un ordonnanceur qui garde la trace d'un sous-ensemble Π' de processus qui n'ont jamais communiqué entre eux car ils proposent toujours les mêmes valeurs dans les objets consensus binaires, et les valeurs qu'ils écrivent dans les registres de fenêtre sont écrasées. La propriété maintenue par les exécutions produites par ce planificateur, appelée Π'-partitionnement, est spécifiée dans la définition 22. Le planificateur construit une exécution à laquelle participent un grand nombre de processus, et de plus en plus d'objets partagés sont couverts par de nombreux processus (c'est-à-dire que ces processus essaient d'écrire dans les objets, voir Définition 23) qui ne connaissent pas l'existence des uns et des autres, jusqu'à ce que tous les objets soient couverts et que deux processus décident de valeurs différentes.

Définition 22 (Exécution partitionnée)**.** *Soit $\Pi' \subset \Pi$ un ensemble de processus, soit $\sim$ une relation d'équivalence sur Π, et soit $p \in \Pi'$ un processus. On dit qu'une exécution finie α est $(\Pi', \sim, p)$-partitionnée (ou simplement Π'-partitionnée si $\sim$ et p sont immatériels) si : 1) pour tous les processus $q, q' \in \Pi'$ $q \nsim q'$, 2) pour tous les processus $q \in \Pi'$, la restriction α_q de α aux étapes effectuées par les processus $q' \sim q$ est une exécution valide de l'algorithme, et 3) tous les registres partagés et les objets consensus ont la même valeur dans $C(\alpha)$ et $C(\alpha_p)$.*

Définition 23 (Configuration couverte)**.** *Soit $\Pi' \subset \Pi$ un ensemble de processus, soit $n \in \mathbb{N}$, soit Y un ensemble d'objets partagés. On dit qu'une configuration C est (Π', n, Y)-couverte si, dans C, pour chaque objet $x \in Y$, 1) x est un registre fenêtré et l'étape suivante d'au moins n processus de Π' est une écriture sur x,*

ou 2) x est un objet de consensus binaire et il existe une valeur commune $v \in \mathbb{B}$ telle que la prochaine étape d'au moins n processus de Π' est de proposer v sur x.

Soit p un processus et x un objet, nous disons que p couvre x dans une configuration C si C est $(\{p\}, 1, \{x\})$-couvert, c'est-à-dire si p est sur le point d'écrire dans x ou de proposer une valeur à x.

Proposition 2. *$M_2[\texttt{cons}\langle\mathbb{B}\rangle, \texttt{WReg}]$ n'est pas universel.*

Démonstration. Supposons qu'il existe un algorithme A qui résolve le consensus dans $M_2[\texttt{cons}\langle\mathbb{B}\rangle, \texttt{WReg}]$. Pour simplifier la preuve, nous supposons également que les processus démarrent l'algorithme en écrivant leur valeur dans un registre first, et terminent en écrivant leur valeur décidée dans un autre registre last. Remarquons que de tels registres et étapes peuvent être ajoutés dans l'algorithme de consensus sans perte de généralité. A l'initialisation de A, un ensemble fini X des objets $m = |X|$ est créé. Nous identifions les registres de lecture/écriture et les objets 1-WReg dans cette preuve. Soit l un entier positif supérieur à la taille de tous les registres fenêtrés dans X, et soit $u_i = (m-i+1)! \times (2l)^{(m-i+1)}$, pour tous les $i \in \{0, ..., m\}$. Nous considérons les exécutions des processus de l'ensemble fini $\Pi = \{p_1, \ldots, p_{u_1}\}$.

Nous construisons, par induction, une séquence $(\Pi_i)_{1 \leq i \leq m}$ d'ensembles de processus, une séquence $(x_i)_{1 \leq i \leq m}$ d'objets partagés de X (on note $X_i = \{x_1, \ldots, x_i\}$) et une séquence croissante $(\alpha_i)_{1 \leq i \leq m}$ d'exécutions Π_i-partitionnées conduisant à une configuration (Π_i, u_i, X_i)-couverte.

Dans l'exécution α_1, chaque processus $p_i \in \Pi$ propose son propre identifiant i et arrête son exécution lorsqu'il est sur le point d'écrire dans $x_1 = \text{first}$. Nous définissons $\Pi_1 = \Pi$ et $\sim_1$ comme l'égalité des processus. L'exécution α_1 est Π_1-partitionnée car aucun processus n'a accédé à un seul objet partagé, et $C(\alpha_1)$ est (Π_1, u_1, X_1)-couvert par construction.

Supposons que nous ayons construit un ensemble de processus Π_i, une séquence d'objets $x_1, \ldots, x_i$ et une exécution Π_i-partitionnée α_i conduisant à une configuration (Π_i, u_i, X_i)-couverte, pour certains $i \in \{0, \ldots, m-1\}$. Soit $v = 2 \times (m-i) \times u_{i+1}$. Nous construisons, par induction, une séquence d'exécutions $\alpha_i^0, \ldots, \alpha_i^v$ et une séquence d'ensembles de processus $\Pi_i^0, \ldots, \Pi_i^v$, de telle sorte que pour tous $j \in \{0, \ldots, v\}$, α_i^j soit $(\Pi_i^j, \sim_i^j, p)$-partitionnée (pour certains $\sim_i^j$ et p), $C(\alpha_i^j)$ est $(\Pi_i^j, (u_i - j), X_i)$-couverte, et un nombre j de processus de Π_i^j couvrent

les objets qui ne sont pas dans X_i. Au départ, posons $\alpha_i^0 = \alpha_i$ et $\Pi_i^0 = \Pi_i$. Supposons que nous ayons construit α_i^j et Π_i^j pour un certain $j < v$. Nous construisons $\alpha_i^{j+1} = \alpha_i^j \beta_i^j \gamma_i^j$ comme suit.

Prenons arbitrairement un ensemble Φ de $l \times i$ processus de Π_i^j, tel que $C(\alpha_i^j)$ est (Φ, l, X_i)-couverte. Φ existe parce que $C(\alpha_i^j)$ est $(\Pi_i^j, (u_i - j), X_i)$-couverte, et $u_i - j > l$. L'extension β_i^j est composée d'une étape de chaque processus $q \in \Phi$. Choisissons arbitrairement un processus $p \in \Phi$, et soit $\Pi' = (\Pi_i^j \setminus \Phi) \cup \{p\}$, et soit $\sim'$ la relation d'équivalence construite en fusionnant les classes d'équivalence des processus dans Φ dans $\sim_i^j$. Comme l est supérieur à la taille de tous les registres fenêtrés, ceux-ci sont remplacés dans β_i^j par des valeurs écrites par des processus de Φ, donc $\alpha_i^j \beta_i^j$ est $(\Pi', \sim', p)$-partitionnée.

Construisons γ_i^j en exécutant p jusqu'à ce qu'il couvre un objet qui n'est pas dans $\{x_1, \dots x_i\}$. Une telle situation doit se produire car, 1) comme A est *wait-free*, p ne peut pas s'exécuter de manière isolée pour toujours, et 2) si p décide d'une valeur v proposée par un processus $p_v \sim_{i+1} p$, alors $l \times i$ autres processus $p' \nsim_i p_v$ peuvent écraser les objets $x_1, \dots x_i$ et décider d'une autre valeur, violant ainsi la propriété de l'accord.

Soit $\alpha_{i+1} = \alpha_i^v$, l'exécution obtenue après avoir répété $v = 2 \times (m-i) \times u_{i+1}$ fois l'ordonnancement précédent, et $\Pi_{i+1} = \Pi_i^v$. Par le Principe des tiroirs, il existe l'un des objets $m - i$ dans $X \setminus X_i$, qui définit x_{i+1}, qui est couvert par au moins $2 \times u_{i+1}$ processus de Π_{i+1}. Si x_{i+1} est un objet de consensus binaire, par le principe des tiroirs à nouveau, la valeur la plus proposée est proposée au moins u_{i+1} fois. De plus, $C(\alpha_{i+1})$ est $(\Pi_{i+1}, (u_i - v), X_i)$-couverte, avec $u_i - v = 2 \times l \times u_{i+1} \geq u_{i+1}$. Par conséquent, $C(\alpha_{i+1})$ est $(\Pi_{i+1}, u_{i+1}, X_{i+1})$-couverte.

Enfin, α_m est $(\Pi_m, \sim, p)$-partitionnée, pour certains $\sim$ et p. Dans $C(\alpha_{i+1})$, au moins $2 \leq u_m$ processus $p_i \neq p_j \in \Pi_m$ sont sur le point d'écrire respectivement v et w, dans le registre last, tel que v a été proposé par le processus p_v et w a été proposé par le processus p_w, avec $p_v \sim p_i \nsim p_j \sim p_w$. Ensuite, p_i et p_j décident d'une valeur différente, ce qui viole la propriété de l'accord. □

Théorème 4. *Le consensus binaire a le numéro de consensus ∞_1^3.*

Démonstration. Comme indiqué précédemment, $M_1[\mathtt{cons}\langle\mathbb{B}\rangle]$ est universel, et par la proposition 1, il en est de même pour $MA_3[\mathtt{cons}\langle\mathbb{B}\rangle]$. Par la proposition 2, $M_2[\mathtt{cons}\langle\mathbb{B}\rangle]$ n'est pas universel. En conclusion, $\mathtt{cons}\langle\mathbb{B}\rangle$ a le numéro de consensus ∞_1^3. □

5.4 Le registre fenêtré a le numéro de consensus ∞_1^2

De par le théorème 3, les objets qui ont le numéro de consensus ∞_1^2 sont les objets les plus faibles qui peuvent être utilisés pour résoudre le consensus entre n processus, pour tout n. Un objet naturel pour remplir ce degré de la hiérarchie est le registres fenêtré, qui peut être considéré comme une composition de registres n-fenêtrés pour tout n, chacun ayant pour numéro de consensus n. Cette section prouve que, en effet, le registre fenêtré a le numéro de consensus ∞_1^2 (Théorème 5). Nous devons d'abord montrer le résultat préliminaire que `WReg` n'est pas universel dans MA_3 (Proposition 3).

Il a déjà été noté dans [AMW11] que l'accès aux objets $\mathsf{cons}^n\langle T\rangle$ pour tous les n n'était pas suffisant pour résoudre le consensus dans MA_3. Cette section adapte les même arguments aux registres fenêtrés. La preuve par l'absurde repose sur une extension des notions classiques de valence aux exécutions qui ne contiennent que des étapes provenant de processus aux identifiants inférieurs à n (Définition 24).

Définition 24 (configuration n-critique)**.** *Soit α une exécution d'un algorithme de consensus. Nous disons que la configuration $C(\alpha)$ est v-n-valente si v peut être décidé dans une extension $\alpha\beta$ de α dans laquelle seuls les processus $p_0, p_1, \dots p_{n-1}$ exécutent des pas de calcul. On dit que $C(\alpha)$ est n-bivalente si elle est à la fois v-n-valente et w-n-valente pour deux valeurs $v \neq w$, et qu'elle est v-n-univalente si elle est v-n-valente et non n-bivalente. Enfin, nous disons que $C(\alpha)$ est n-critique si elle est n-bivalente et que la prochaine étape prise par tout processus dans $p_0, p_1, \dots p_{n-1}$ conduit à une configuration v-n-univalente, pour un certain v.*

Lemme 18. *Toute exécution finie α telle que $C(\alpha)$ est n-bivalente a une extension $\alpha\beta$ telle que $C(\alpha\beta)$ est n-critique.*

Démonstration. Supposons que ce ne soit pas le cas. Nous construisons une exécution infinie $\alpha' = \alpha\beta_1\beta_2\dots$ de telle sorte que, pour tout i, $\alpha_i = \alpha\beta_1\beta_2\dots\beta_i$ conduit à une configuration n-bivalente. Pour $i = 0$, $\alpha_0 = \alpha$ conduit à une configuration n-bivalent. Supposons que nous ayons construit une telle exécution pour quelque i. Par hypothèse, $C(\alpha_i)$ n'est pas n-critique, il existe donc un processus p_i dont l'étape suivante est β_{i+1} de telle sorte que $\alpha_{i+1} = \alpha_i\beta_{i+1}$ mène à une configuration

n-bivalente. Enfin, α' est infinie mais un nombre fini de processus est arrivé, donc certains processus ont pris un nombre infini d'étapes, ce qui contredit la propriété de *wait-freedom*. □

Lemme 19. *Dans toutes les configurations critiques n, $p_0, \ldots, p_{n-1}$ sont sur le point d'écrire dans le même objet k-`WReg`, avec $k \geq n$.*

Démonstration. Soit α une exécution menant à une configuration n-critique. Chaque processus p_i est sur le point d'exécuter une étape β_i sur un objet partagé.

Supposons qu'il existe un processus p_i dont la prochaine étape est une lecture. Comme $C(\alpha)$ est critique, $C(\alpha\beta_i)$ est v-n-univalente et il existe p_j tel que $C(\alpha\beta_j)$ est w-n-univalente, avec $v \neq w$. C'est impossible puisque $C(\alpha\beta_i\beta_j)$ est v-n-univalente, mais $C(\alpha\beta_j)$ et $C(\alpha\beta_i\beta_j)$ sont indiscernables du point de vue de p_j.

Supposons que les processus soient sur le point d'écrire dans au moins deux registres fenêtrés différents. Comme $C(\alpha)$ est critique, il existe deux processus p_i et p_j écrivant dans des registres fenêtrés différents tels que $C(\alpha\beta_i)$ est v-n-univalente et $C(\alpha\beta_j)$ est w-n-univalente, avec $v \neq w$. Ceci est impossible car $C(\alpha\beta_i\beta_j)$ est v-n-univalente, $C(\alpha\beta_j\beta_i)$ est w-n-univalente, et $C(\alpha\beta_i\beta_j) = C(\alpha\beta_j\beta_i)$.

Supposons que les processus $p_0, \ldots, p_{n-1}$ soient sur le point d'écrire dans le même objet k-`WReg`, avec $k < n$. Soit p_i et p_j tels que $C(\alpha\beta_i)$ est v-n-univalente et $C(\alpha\beta_j)$ est w-n-univalente, avec $v \neq w$. Soit γ une concaténation des étapes de tous les processus, sauf p_i et p_j. $C(\alpha\beta_i\beta_j\gamma)$ est v-n-univalente et $C(\alpha\beta_j\gamma)$ est w-n-univalente, mais les deux configurations sont indiscernables par p_j.

Le seul cas restant est celui où les processus $p_0, \ldots, p_{n-1}$ sont sur le point d'écrire dans le même objet k-`WReg`, avec $k \geq n$. □

Proposition 3. *MA_3[`WReg`] n'est pas universel.*

Démonstration. Supposons qu'il existe un algorithme A qui résout le consensus dans MA_3[`WReg`]. Nous construisons une séquence d'exécutions $\alpha_0 = \beta_0, \alpha_1 = \beta_0\beta_1, \alpha_2 = \beta_0\beta_1\beta_2, \ldots$ et une séquence d'entiers $n_0 \leq n_1 \leq n_2 \leq \ldots$ tels que, pour tout i, le processus p_0 fait un pas dans β_i et $C(\alpha_i)$ est n_i-critique.

Pour $i = 0$, soit $n_0 = 2$ et soit γ l'exécution dans laquelle p_0 et p_1 proposent respectivement 0 et 1. Dans une extension p_0-solo de γ, p_0 décide 0, et dans une extension p_1-solo de γ, p_1 décide 1, donc $C(\gamma)$ est n_0-bivalente. De par le lemme 18, il y a une extension α_0 de γ telle que $C(\alpha_0)$ est n_0-critique.

Supposons que nous ayons construit une exécution α_i et un entier n_i respectant l'invariant d'induction pour un certain i. De par le lemme 19, dans $C(\alpha_i)$, $p_0, \ldots, p_{n_i-1}$ sont sur le point d'écrire dans le même objet k-`WReg`, avec $k \geq n_i$. Posons $n_{i+1} = k+1$. Comme $C(\alpha_i)$ est n_i-critique, $C(\alpha_i)$ est également n_i-bivalente, donc $C(\alpha_i)$ est n_{i+1}-bivalente. Par le lemme 18, il y a une extension $\alpha_{i+1} = \alpha_i\beta_{i+1}$ de α_i telle que $C(\alpha_{i+1})$ est n_{i+1}-critique. Par le Lemme 19, dans $C(\alpha_{i+1})$, p_0 est sur le point d'écrire dans un objet k'-`WReg`, avec $k' > k$. En particulier, p_0 a effectué son étape d'écriture dans β_{i+1}.

Pour conclure, p_0 a exécuté un nombre infini d'étapes dans $\alpha = \beta_1\beta_2\ldots$, c'est-à-dire que α n'est pas *wait-free*. C'est une contradiction. □

Théorème 5. *Le registre fenêtré a pour numéro de consensus* ∞_1^2.

Démonstration. Par [MPR18], n-`WReg` a le numéro de consensus n. Donc, M_1^n[`WReg`] est universel pour tout n, donc M_1[`WReg`] est universel. Par le théorème 3, `WReg` a au moins le numéro de consensus ∞_1^2. Par la proposition 3, MA_3[`WReg`] n'est pas universel et par la proposition 2, M_2[`WReg`] n'est pas universel. Par conséquent, `WReg` a le numéro de consensus ∞_1^2. □

5.5 Objets ayant pour numéro de consensus ∞_2^3

Comme un objet avec le numéro de consensus ∞_1^3 est universel dans MA_3 et un objet avec le numéro de consensus ∞_2^2 est universel dans M_2, leur composition ne peut avoir que le numéro de consensus ∞_2^3 ou ∞_3^3. Dans cette section, nous prouvons que la composition du consensus binaire et des piles d'itérateurs, nos exemples respectifs de numéros de consensus ∞_1^3 et ∞_2^2, n'est pas universelle dans M_3 (Proposition 4), elle a donc le numéro de consensus ∞_2^3 (Théorème 6).

Comme pour la proposition 2, la preuve de la proposition 4 construit un ordonnanceur qui construit une exécution Π'-partitionnée , en gardant une trace d'un sous-ensemble Π' de processus qui n'ont jamais communiqué entre eux et dans lesquels de plus en plus d'objets partagés sont couverts (la définition 25 adapte la notion de couverture pour prendre les piles d'itérateurs en compte). La principale difficulté est que les piles d'itérateurs ne peuvent pas être écrasées par un nombre fini de processus, et la preuve basée sur la valence introduite dans [AMW11] ne peut pas être adaptée à un environnement où les objets de consensus binaire peuvent être utilisés dans une configuration critique. Le lemme 20 permet

à l'ordonnanceur d'introduire un flux de processus nouvellement arrivés qui, en couvrant, lisant ou écrivant toutes les piles d'itérateurs, empêche n'importe quel processus choisi essayant d'accéder à une pile d'itérateurs d'apprendre de l'information sur l'existence d'autres processus. Cette intuition est spécifiée dans la définition 26, par le concept d'extensions aveugles.

Définition 25 (Configuration couverte)**.** *Un objet x est couvert en écriture par un processus p dans une configuration C si : 1) x est un registre et l'étape suivante de p dans C est une écriture dans x, 2) x est un objet de consensus binaire et la prochaine étape de p dans C est la proposition d'une valeur dans x, ou 3) x est une pile d'itérateurs et la prochaine étape de p dans C est une écriture sur x.*

Un objet x est couvert par un processus p dans une configuration C si x est couvert en écriture, ou si x est une pile d'itérateurs et l'étape suivante de p dans C est une lecture sur x.

Soit $\Pi' \subset \Pi$ un ensemble de processus, soit $n \in \mathbb{N}$, soit Y un ensemble d'objets partagés. On dit qu'une configuration C est (Π', n, Y)-couverte si, dans C, tous les objets de Y sont couverts au moins n fois par des processus de Π', et, dans le cas d'un objet consensus binaire x, au moins n processus sont sur le point de proposer la même valeur.

Définition 26 (Extension aveugle)**.** *Soit α une exécution $(\Pi', \sim, p)-$partitionnée. Nous disons que $\alpha\beta$ est une extension aveugle de α si : 1) aucun processus n'a effectué de pas de calcul à la fois dans α et β, et 2) pour chaque processus q exécutant des étapes dans β, il y a une extension $\alpha_p\beta'$ de α_p [1] telle que l'état local de q est le même dans $C(\alpha\beta)$ et dans $C(\alpha_p\beta')$. En d'autres termes, seuls les nouveaux processus ont effectué des pas de calculs dans β, mais ils n'ont pas pu apprendre l'existence de processus autres que ceux équivalents à p.*

Lemme 20. *Soit α une exécution $(\Pi', \sim, p)$-partitionnée d'un algorithme de consensus A, soit X l'ensemble des objets instanciés lors de la configuration de A, et soit $m = |X|$.*

Pour tous les $k \in \{0, \ldots, m\}$, il existe une extension aveugle $\alpha\beta$ de α telle que au moins k objets différents sont couverts en écriture dans $C(\alpha\beta)$ par les processus $r_1, \cdots, r_k$ qui n'ont pas pris de pas de calcul dans α.

1. Rappel : α_p est la restriction de α aux opérations des processus de la classe d'équivalence de p selon $\sim$.

Démonstration. Nous prouvons cette proposition par induction sur k. Pour $k = 0$, nous posons $\beta = \varepsilon$, l'exécution vide. Supposons, comme hypothèse d'induction $H(k)$, que la proposition soit valable pour $k \in \{0, \ldots, m-1\}$. Nous commençons la preuve de $H(k+1)$ en prouvant une autre proposition.

Déclaration 3. *Il existe une extension aveugle $\alpha\beta$ de α telle que au moins k objets différents sont couverts en écriture dans $C(\alpha\beta)$ par les processus $r_1, ..., r_k$ qui n'ont pas effectué d'étapes dans α, et un autre objet est couvert dans $C(\alpha\beta)$ par un processus r_{k+1} qui n'a pas effectué d'étapes dans α.*

Démonstration. Supposons que cette affirmation soit fausse. Nous construisons une exécution infinie $\alpha\beta_0\beta_1\beta_2\ldots$ dans laquelle un processus prend un nombre infini d'étapes, de sorte que chaque extension $\alpha\beta_0\ldots\beta_n$ soit aveugle. Soit w le nombre d'écritures sur les piles d'itérateurs dans α, soit $\alpha\beta_0 = \alpha\gamma_1\ldots\gamma_{w+2}$ l'exécution aveugle obtenue après l'invocation de l'hypothèse d'induction $H(k)$, $w+2$ fois, et soit Y_l l'ensemble contenant les objets k couverts en écriture dans γ_l, pour chaque l. Comme nous supposons que la proposition 3 est fausse, $Y = \bigcup_{l=1}^{w+2} Y_l$ a la taille k et chaque objet $y \in Y$ est couvert en écriture au moins $w+2$ fois dans $C(\alpha\beta_0)$. Soit p un processus qui n'a pas effectué d'étapes dans $\alpha\beta_0$. Supposons que nous ayons construit $\delta_n = \alpha\beta_0\ldots\beta_n$. Nous construisons β_{n+1} comme suit.

— Supposons que p soit sur le point de lire une pile d'itérateurs $y \in Y$ dans la configuration $C(\delta_n)$. Soit w' le nombre d'écritures sur une pile d'itérateurs dans δ_n. Nous construisons $\delta_{n+1} = \delta_n\zeta_1\ldots\zeta_{w'}\eta$ comme suit : chaque ζ_l est le résultat d'une invocation de l'hypothèse d'induction $H(k)$. Comme nous supposons que la proposition 3 est fausse, l'ensemble des objets couverts en écriture dans chaque ζ_l est Y. En particulier, dans $C(\delta_n\zeta_1\ldots\zeta_{w'})$, y est couvert en écriture w' fois par des processus qui n'ont pas effectué d'étapes de calcul dans δ_n. Dans η, nous laissons les processus w' écrire dans y, p lit alors dans y et obtient l'une des valeurs écrites par l'un de ces processus, ce qui garantit que l'extension est aveugle.

— Si p est sur le point d'écrire dans une pile d'itérateurs $y \in Y$ dans la configuration $C(\delta_n)$, β_{n+1} est uniquement composée de la prochaine étape de p. L'écriture retourne un itérateur $i = i_\alpha + i'$, où i_α est le nombre d'écritures sur y dans α et i' est le nombre d'écritures sur y dans $\beta_0\ldots\beta_n$. Comme $i_\alpha \leq w$, p ne peut pas distinguer la valeur de retour avec une

valeur de retour qu'il aurait eu si son écriture dans y avait été précédée de i_α écritures de processus arrivés dans β_0, donc l'extension est aveugle.

— Sinon, dans la configuration $C(\delta_n)$, p est sur le point d'exécuter une étape locale, de lire dans un registre $x \in X$, d'écrire dans un registre $y \in Y$, de proposer une valeur sur un objet consensus $y \in Y$, ou d'accéder à un objet instancié pendant $\beta_0 \dots \beta_n$ ou dans α par un processus $p' \sim p$. Dans tous ces cas, β_{n+1} est uniquement composé de la prochaine étape de p, ce qui est une extension aveugle de δ_n.

En supposant que la proposition 3 est fausse, nous avons construit une exécution dans laquelle le processus p exécute un nombre infini d'étapes, ce qui contredit la propriété de *wait-freedom* et conclut la preuve. □

Continuons la preuve du lemme 20 en supposant que $H(k+1)$ est fausse. Nous construisons une exécution infinie $\alpha\beta_0\beta_1\beta_2\dots$ dans laquelle un certain processus prend un nombre infini de pas de calcul, et tel que chaque extension $\alpha\beta_0\dots\beta_n$ est aveugle.

Soit w le nombre d'opérations d'écriture sur les piles d'itérateurs dans α, et $w' = (m-k)(w^2+1)$. Remarquons que w' est une borne supérieure sur le nombre d'opérations de lecture pouvant renvoyer une valeur non-$\perp$ depuis $(m-k)$ piles d'itérateurs, à partir de $C(\alpha)$. Nous construisons $\alpha\beta_0 = \alpha\gamma_1\dots\gamma_{w'+1}$ de telle sorte que chaque γ_l soit l'extension aveugle donnée par la proposition 3. Comme nous supposons que $H(k+1)$ est fausse, 1) un ensemble Y d'objets k sont couverts $(w'+1)$ fois en écriture dans $C(\alpha\beta_0)$ par des processus arrivés dans β_0, 2) aucun processus n'a écrit dans une pile d'itérateurs $y \notin Y$ dans β_0, et 3) $(w'+1)$ processus qui n'ont pas exécuté d'étapes dans α sont sur le point de lire depuis des piles d'itérateurs qui ne sont pas dans Y. Soit Φ l'ensemble de ces $(w'+1)$ processus.

Supposons que nous ayons construit une extension aveugle $\delta_n = \alpha\beta_0\dots\beta_n$ de α de telle sorte qu'un certain processus dans Φ ait exécuté au moins une étape dans β_l, pour chaque $l \leq n$. Pour construire β_{n+1}, nous choisissons un processus $p \in \Phi$ qui n'a pas lu de valeur $v \neq \perp$ depuis une pile d'itérateurs $y \notin Y$ dans δ_n. Un tel processus existe parce que 1) l'hypothèse que $H(k+1)$ est fausse implique qu'aucun processus n'écrit dans une pile d'itérateurs $y \notin Y$ dans $\beta_0\dots\beta_n$, et 2) il est impossible de lire une valeur autre que $\perp$ dans une pile d'itérateurs $y \notin Y$ plus de $w' < |\Phi|$ fois.

— Supposons que p soit sur le point de lire à partir d'une pile d'itérateurs $y \in Y$ dans la configuration $C(\delta_n)$. Soit w'' le nombre d'écritures sur des piles d'itérateurs dans δ_n, et construisons $\delta_{n+1} = \delta_n\zeta_1 \dots \zeta_{2w''}\eta$ comme suit. Soit $\lambda_1 = \delta_n$ et $\lambda_i = \delta_n\zeta_1 \dots \zeta_{i-1}$ pour $i > 1$. Pour chaque $i \in \{1, \dots, 2w''\}$, $\lambda_i\zeta_i$ est l'extension aveugle la plus courte de λ_i telle qu'un processus qui n'a pas effectué d'étape dans λ_i soit sur le point d'écrire dans y, ou de lire à partir de y dans le même itérateur que p. Une telle extension existe de par la proposition 3 et la supposition que $H(k+1)$ est fausse. Par le principe des tiroirs, deux cas sont possibles dans $C(\delta_n\zeta_1 \dots \zeta_{2w''})$. Si au moins w'' nouveaux processus sont sur le point de lire y, alors η contient leur lecture, ainsi que la lecture par p renvoyant $\perp$. Sinon, au moins w'' nouveaux processus sont sur le point d'écrire dans y, et η contient leur écriture et la lecture de p, qui renvoie une valeur écrite dans η. Dans les deux cas, δ_{n+1} est aveugle.

— Si p est sur le point d'écrire dans une pile d'itérateurs $y \in Y$ dans la configuration $C(\delta_n)$, la seule étape de β_{n+1} est l'opération d'écriture de p. Comme décrit dans la proposition 3, le fait que y soit couvert en écriture au moins $(w''+1)$ fois par les processus arrivés dans β_0 -ce qui est plus que le nombre d'écritures sur y dans α- implique que l'extension est aveugle.

— Dans les autres cas, dans la configuration $C(\delta_n)$, le processus p est sur le point d'exécuter une étape locale, de lire dans un registre $x \in X$, d'écrire dans un registre $y \in Y$, de proposer une valeur sur un objet consensus $y \in Y$, ou d'accéder à un objet instancié pendant $\beta_0 \dots \beta_n$ ou dans α par un processus $p' \sim p$. Dans tous ces cas, β_{n+1} est uniquement composé de la prochaine étape de p, ce qui est une extension aveugle de δ_n.

En supposant que $H(k+1)$ était fausse, nous avons construit une exécution dans laquelle certains processus exécutent un nombre infini d'étapes, ce qui contredit la propriété de *wait-freedom* et conclut la preuve. □

Proposition 4. *$M_3[\texttt{cons}\langle\mathbb{B}\rangle, \texttt{IStack}]$ n'est pas universel.*

Démonstration. Supposons qu'il existe un algorithme A qui résolve le consensus dans $M_3[\texttt{cons}\langle\mathbb{B}\rangle, \texttt{IStack}]$. De façon similaire à la proposition 2, nous supposons que les processus démarrent l'algorithme en écrivant leur valeur dans un registre first, et terminent en écrivant leur valeur de décision dans un autre registre last. Nous considérons une exécution dans laquelle $\Pi = \{p_0, p_1, p_2, \dots\}$ est

infini et dans laquelle chaque processus p_i propose son identifiant i comme valeur de consensus. A l'initialisation de A, un ensemble fini X de $|X| = m$ objets est créé. Soit $u_i = (m-i+1)!2^{m-i+1}$, pour tout $i \in \{0, ..., m\}$. Nous construisons, par induction, une séquence $(\Pi_i)_{1 \leq i \leq m}$ d'ensembles de processus, une séquence $(x_i)_{1 \leq i \leq m}$ d'objets partagés de X (on note $X_i = \{x_1, \ldots, x_i\}$) et une séquence croissante $(\alpha_i)_{1 \leq i \leq m}$ d'exécutions Π_i-partitionnées conduisant à une configuration (Π_i, u_i, X_i)-couverte.

Nous posons $\Pi_1 = \Pi$ et $x_1 = \mathsf{first}$. Dans l'exécution α_1, chaque processus $p_j \in \{p_0, \ldots, p_{u_1}\}$ propose son identifiant j et arrête de s'exécuter au moment d'écrire dans first. Comme aucune opération sur les objets partagés ne s'est produite dans α_1, α_1 est Π_1-partitionnée et $C(\alpha)$ est (Π_1, u_1, X_1)-couverte.

Supposons que nous ayons construit un ensemble de processus Π_i, un ensemble d'objets X_i et une exécution Π_i-partitionnée α_i conduisant à une configuration (Π_i, u_i, X_i)-couverte, pour un certain $i < m$. Soit $v = 2 \times (m-i) \times u_{i+1}$. Nous construisons, par induction, une séquence d'exécutions $\alpha_i^0, \ldots, \alpha_i^v$ et une séquence d'ensembles de processus $\Pi_i^0, \ldots, \Pi_i^v$, de telle sorte que pour tout $j \in \{0, \ldots, v\}$, α_i^j soit $(\Pi_i^j, \sim_i^j, p)$-partitionnée (pour certains $\sim_i^j$ et p), $C(\alpha_i^j)$ soit $(\Pi_i^j, (u_i - j), X_i)$-couverte, et que j processus de Π_i^j couvrent des objets qui ne sont pas dans X_i. Au départ, posons $\alpha_i^0 = \alpha_i$ et $\Pi_i^0 = \Pi_i$. Supposons que nous ayons construit α_i^j et Π_i^j pour un certain $j < v$. Nous construisons $\alpha_i^{j+1} = \alpha_i^j \beta_i^j \gamma_i^j$ comme suit. Prenons quelques processus $q_1, \ldots, q_i \in \Pi_i^j$ couvrant respectivement $x_1, \ldots, x_i$ dans $C(\alpha_i^j)$.

L'extension β_i^j est composée d'une étape de chaque processus $q_1, \ldots, q_i$. Soit $\Pi' = \Pi_i^j \setminus \{q_2, \ldots, q_i\}$ et soit $\sim'$ la relation d'équivalence construite en fusionnant les classes d'équivalence de $q_1, \ldots, q_i$ dans $\sim_i^j$. L'exécution $\alpha_i^j \beta_i^j$ est $(\Pi', \sim', q_1)$-partitionnée.

Soit $\alpha_i^j \beta_i^j \gamma_i^j$ l'extension aveugle la plus courte de $\alpha_i^j \beta_i^j$ telle que, dans $C(\alpha_i^j \beta_i^j \gamma_i^j)$, un processus p_i^j couvre un objet $y_i^j \notin X_i$. Une telle extension existe de par le lemme 20 pour $k = i+1$, et comme nous avons considéré la plus courte extension de ce type, seuls les objets de X_i ont un changement de leur état.

Posons $\alpha_{i+1} = \alpha_i^v$. Au moins $v = 2 \times (m-i) \times u_{i+1}$ objets sont couverts dans α_{i+1}. Par le principe des tiroirs, il existe l'un des $m-k$ objets qui n'est pas dans X_i, noté x_{i+1}, qui est couvert par au moins $2 \times u_{i+1}$ processus. Si x_{i+1} est un objet de consensus binaire, par le principe des tiroirs à nouveau, la valeur

la plus proposée est proposée au moins u_{i+1} fois. Notons Φ_{i+1} l'ensemble de ces processus. De plus, au moins $u_i - v = 2u_{i+1} > u_{i+1}$ processus couvrent toujours chaque objet de X_i. Nous construisons l'ensemble Π_{i+1} comme l'union de Φ_{i+1} et de l'ensemble des processus qui n'ont pas exécuté d'étape dans $\alpha_i^1, \ldots, \alpha_i^v$. L'exécution α_{i+1} est Π_{i+1}-partitionnée et $C(\alpha_{i+1})$ est $(\Pi_{i+1}, u_{i+1}, X_{i+1})$-couverte, ce qui conclut l'induction.

Enfin, α_m est $(\Pi_m, \sim, p)$-partitionnée, pour certains $\sim$ et p. Dans $C(\alpha_{i+1})$, au moins $u_m = 2$ processus $p_i \neq p_j \in \Pi_m$ sont sur le point d'écrire respectivement v et w, dans le registre last, tels que v ait été proposée par le processus p_v et w ait été proposée par le processus p_w, avec $p_v \sim p_i \nsim p_j \sim p_w$. Ensuite, p_i et p_j décident d'une valeur différente, ce qui viole la propriété de l'accord. □

Théorème 6. *La composition des piles d'itérateurs et du consensus binaire a le numéro de consensus* ∞_2^3.

Démonstration. De par [AMW11], $M_2[\texttt{IStack}]$ est universel, et par la proposition 1, $MA_3[\texttt{cons}\langle\mathbb{B}\rangle]$ est universel, donc $\texttt{IStack} + \texttt{cons}\langle\mathbb{B}\rangle$ a au moins le numéro de consensus ∞_2^3. Par la proposition 4, $\texttt{IStack} + \texttt{cons}\langle\mathbb{B}\rangle$ a au plus le numéro de consensus ∞_2^3. □

5.6 Conclusion

Ce chapitre explore l'universalité des objets partagés dans les modèle ouverts où il n'est pas possible d'allouer et d'initialiser, en une fois, un nombre infini d'emplacements de mémoire. Nous avons prolongé la hiérarchie des systèmes *wait-free* existante en séparant les objets avec un numéro de consensus infini en 5 catégories, en fonction de leur universalité dans les modèles d'arrivées bornées, finies ou infinies, et la nécessité ou non d'avoir un mécanisme d'allocation de mémoire infinie. Ce chapitre soulève plusieurs nouvelles questions ouvertes, que nous détaillerons par la suite.

Dans ce chapitre, nous supposons que les processus partagent une mémoire infinie. Bien que cette hypothèse soit au cœur de la définition de la machine de Turing à la base de l'informatique, cela implique naturellement que les pointeurs vers les emplacements mémoire aient une taille infinie, ce qui est moins pratique. Sans cette hypothèse, un consensus à valeurs multiples pourrait être

résolu en utilisant un nombre d'objets de type consensus binaires égal à la taille d'un pointeur [ZC09]. Un problème ouvert intéressant est l'existence ou non d'un objet partagé avec un numéro de consensus ∞_1^3 qui n'a pas d'implémentation polylogarithmique du consensus dans MA_2.

Les objets que nous étudions dans ce chapitre, en particulier les registres fenêtrés et les piles d'itérateurs, sont complexes dans le sens où ils nécessitent généralement plusieurs emplacements de mémoire pour être mis en œuvre. Un autre problème ouvert est l'existence d'instructions spécifiques qui fonctionnent sur un seul (ou un nombre fixe et petit) emplacement de mémoire, qui remplissent chaque degré de la nouvelle hiérarchie.

Enfin, l'exemple d'objet de numéro de consensus ∞_2^3 que nous avons présenté dans ce chapitre est une composition d'un objet avec le numéro de consensus ∞_2^2 et d'un objet avec le numéro de consensus ∞_1^3. Il serait intéressant de rechercher si c'est toujours le cas, ce qui peut être divisé en deux questions :

1) Existe-t-il un objet de numéro de consensus ∞_2^3 qui ne peut pas être exprimé comme une telle composition ? 2) Inversement, existe-t-il deux objets de consensus numéro ∞_2^2 et ∞_1^3 dont la composition a le numéro de consensus ∞_3^3 ?

Chapitre 6

Constructions universelles faiblement cohérentes dans un système par passage de message : études des complexités spatiales et de communication [BMP19a]

Énoncé du problème Ce chapitre explore la question suivante : quelle est la complexité des constructions universelles *update* cohérentes dans les systèmes sujets aux partitions ? Deux métriques sont pertinentes : le nombre de messages envoyés pour chaque opération, et la taille des métadonnées stockées par chaque processus.

Du point de vue de la calculabilité, des constructions universelles *update* cohérentes ont déjà été proposées dans le modèle opérationnel [PMJ15 ; PMJ16]. L'idée clé de ces algorithmes est d'ordonner les opérations *a priori* en utilisant un *timestamp* donné lors de la phase de préparation, et de tenir un journal de toutes les opérations ordonnées en fonction de cet horodatage dans la phase de mise à jour. Lorsque deux processus ont reçu le même ensemble de messages, ils s'accordent sur leur journal respectif, de sorte qu'ils convergent également vers le même état. Le principal problème de cette approche est la taille du journal, qui ne cesse d'augmenter à mesure que des opérations sont appelées. On peut affirmer que même si le système est asynchrone, il est très peu probable que le délai de transfert d'un message dépasse par exemple une journée et par conséquent, tous les états « très anciens » peuvent être supprimés car aucun ancien message

ne peut forcer à ré-exécuter les nouvelles opérations. Cela peut fonctionner mais n'est pas sûr, car si, pour une raison quelconque, une transmission de message dépasse ce délai maximal supposé, la convergence n'est plus garantie. D'où les questions suivantes : est-il possible de retirer en toute sécurité les opérations très anciennes du journal, et, si oui, à quel prix ?

Contributions du chapitre Ce chapitre a deux contributions principales.

— Ce chapitre prouve d'abord que la réponse est « non » dans le modèle opérationnel, mais « oui » dans le modèle *wait-free*. Pour cela, nous introduisons un objet appelé l-countdown-append, où l est un paramètre entier. Nous prouvons que $\Omega(l)$ bits de mémoire locale sont nécessaires dans le modèle opérationnel pour implémenter un objet l-countdown-append avec la cohérence d'écriture, mais nous donnons un algorithme dont la complexité est logarithmique dans le modèle *wait-free*. Cette contribution a des implications théoriques profondes, car elle prouve que les deux modèles ne sont pas équivalents en termes de complexité, ce qui questionne la pertinence de résultats de complexité déjà connus concernant les algorithmes CRDT.

— Malheureusement, l'algorithme logarithmique présenté dans la première contribution pour le modèle *wait-free* nécessite $O(n^2)$ messages par opération. Nous proposons un compromis pratique, une construction universelle appelée $UC[k]$, qui ne nécessite que n messages par opération lorsqu'il y a une borne k sur la concurrence et fournit un mécanisme de *garbage collection* pour les anciens messages qui est sûr même lorsque la limite est dépassée. Le paramètre k rend compte de la relation entre le délai de transmission relatif des messages et le nombre d'opérations de mise à jour émises pendant la transmission d'un message.

Organisation du chapitre Le reste de ce chapitre est organisé comme suit. La section 6.1 rappelle la construction universelle UC_∞ dans le modèle opérationnel, qui a été introduit dans [PMJ15]. La section 6.2 prouve la borne inférieure dans le modèle opérationnel et donne une borne supérieure dans le modèle *wait-free*, prouvant que les deux modèles ne sont pas équivalents quand on considère la complexité. La section 6.3 présente enfin la construction universelle $UC[k]$. Puis, la section 6.4 conclut le chapitre.

```
1 Opération apply(o ∈ O) ∈ R
2     var s ∈ S ← s_0;
3     if o is a query then
4         for (t, j, o') ∈ history_i sorted according to (t, j) do s ← u(s, o') ;
5     if o is an update then time_i ← time_i + 1 ; broadcast mUpdate
        (time_i, o) ;
6     return q(s, o);
7 Quand un message mUpdate (t_j ∈ ℕ, o_j ∈ O) est reçu depuis p_j
8     time_i ← max(time_i, t_j) ;
9     history_i ← history_i ∪ {(t_j, j, o_j)};
```

Algorithme 7 : Construction universelle $UC_\infty(\mathcal{O}, \mathcal{R}, \mathcal{S}, s_0, u, q)$: code pour p_i

6.1 Une construction universelle dans le modèle opérationnel

Nous rappelons maintenant le fonctionnement de la construction universelle *update* cohérente dans le modèle opérationnel, qui a été introduite dans [PMJ15]. Le code de l'algorithme 7 est donné pour le processus p_i.

Le principe est de construire un ordre total sur les opérations de mises à jour sur lequel tous les participants sont d'accord *a priori*, puis de réécrire l'historique *a posteriori* afin que chaque réplicat de l'objet atteigne finalement l'état correspondant à l'histoire séquentielle commune. Dans l'algorithme 7, cet ordre est construit à partir d'une horloge de Lamport [Lam78] qui contient la relation *happened-before*. Afin d'avoir un ordre total, les événements sont horodatés avec une paire composée de l'horloge logique et de l'identifiant du processus qui l'a produite.

Chaque processus p_i gère sa vue time_i de l'horloge logique et une liste $\mathsf{history}_i$ de tous les événements de mise à jour horodatés dont le processus p_i est au courant. La liste $\mathsf{history}_i$ contient des triplets (t, j, o) où o est une opération de mise à jour et (t, j) l'horodatage. Cette liste est triée en fonction des horodatages des mises à jour : $(t, j) < (t', j')$ si $(t < t')$ ou $(t = t'$ et $j < j')$.

Lorsqu'une mise à jour est émise localement, le processus p_i l'horodate et informe tous les autres processus en diffusant (de manière fiable) un message à tous les autres processus (y compris lui-même), à la ligne 5. Par conséquent, tous

les processus seront éventuellement informés de toutes les mises à jour. Lorsqu'un message $\texttt{mUpdate}(t_j, o_j)$ est reçu, p_i met à jour son horloge et insère l'événement dans la liste $\mathsf{history}_i$ (Ligne 7). Lorsqu'une requête est émise, p_i rejoue localement toute la liste des événements de mises à jour dont il a conscience à partir de l'état initial, puis il exécute la requête sur l'état qu'il vient d'obtenir (Ligne 4).

Dans une exécution qui ne contient qu'un nombre fini d'opérations de mise à jour, alors tous les processus connaîtront finalement le même ensemble de mises à jour et les trieront dans le même ordre afin d'évaluer leurs opérations de requête. Cela implique la cohérence d'écriture. Chaque fois qu'une opération est émise, elle termine sans attendre aucun autre processus. Cela correspond à des exécutions *wait-free* dans des systèmes distribués à mémoire partagée et implique la tolérance aux pannes.

6.2 Comparaison des modèles de calcul

Les algorithmes du modèle opérationnel sont naturellement tolérants aux partitions et, comme un seul message est diffusé par opération de mise à jour, ils sont par conception optimaux en termes de nombre de messages envoyés. Cependant, comme mentionné dans le chapitre 2, le modèle opérationnel impose des limitations sur la forme de ses algorithmes admissibles.

Suite à notre quête d'une construction universelle efficace, nous étudions l'impact de ces limitations sur la complexité en mémoire des algorithmes distribués. En effet, la quantité de métadonnées qui doit être stockée par chaque processus pour assurer la convergence dans le modèle opérationnel est problématique et a été largement étudié pour plusieurs objets comprenant des ensembles, des compteurs et des registres [Bur+14], des bases de données [AEM17] et des éditeurs collaboratifs [Att+16].

Dans cette section, nous montrons que le modèle opérationnel n'est pas équivalent au modèle *wait-free* en termes de complexité de la mémoire. Plus précisément, nous exposons une famille d'objets, appelés objets *l-countdown-append*, et nous prouvons que tout algorithme *update* cohérent les implémentant dans le modèle opérationnel nécessite au moins $\Omega(l)$ bits de métadonnées dans certaines exécutions spécifiques. D'autre part, seul un nombre logarithmique de bits est requis dans le modèle *wait-free*.

Ce résultat a un impact sur la complexité des constructions universelles. La limite inférieure linéaire prouvée pour l'objet l-countdown-append dans le modèle opérationnel est similaire à la borne supérieure donnée sur les constructions universelles par l'algorithme 7. En particulier, il suggère qu'il est impossible de supprimer en toute sécurité des opérations très anciennes du journal, tout en restant dans le modèle opérationnel. Ceci est néanmoins possible dans le modèle *wait-free*, au prix d'un nombre accru de messages.

D'un point de vue théorique, ce résultat interroge également la généralité des études de complexité des différents CRDT mentionnés ci-dessus, puisque ces études ne prennent en compte que des algorithmes dans le modèle opérationnel.

Complexité du l-countdown-append *update* cohérent. Notre preuve de non-équivalence compare le nombre de bits nécessaires pour encoder les états locaux des processus lors de l'exécution d'un schéma d'exécutions spécifique sur une famille spécifique d'objets. Il faut d'abord préciser notre notion de complexité et définir les objets *l-countdown-append*.

Complexité des algorithmes déterministes Les exécutions de différents algorithmes dans le modèle *wait-free* peuvent être très différentes, donc nous ne pouvons comparer que des algorithmes en se basant sur les histoires distribuées, qui forment la base de leur spécification. Nous définissons la *H-complexité* comme la taille maximale d'un état local accessible lors d'une exécution abstraite par l'historique H.

Définition 27 (H-complexité)**.** *Soit H un historique contenant un nombre fini d'opérations de mises à jour, et A un algorithme. Soit également S l'ensemble de tous les états locaux accessibles par tout processus exécutant A pendant une exécution qui peut être abstraite par H.*

Nous définissons la H-complexité de A comme suit :

- *si $S = \emptyset$ (c'est-à-dire si H n'est pas admis par A), la complexité de H est 0 ;*
- *si S est infini (c'est-à-dire si S contient des états de taille non bornée), la complexité de H est ∞ ;*
- *sinon, la complexité H est la taille maximale d'un état dans S.*

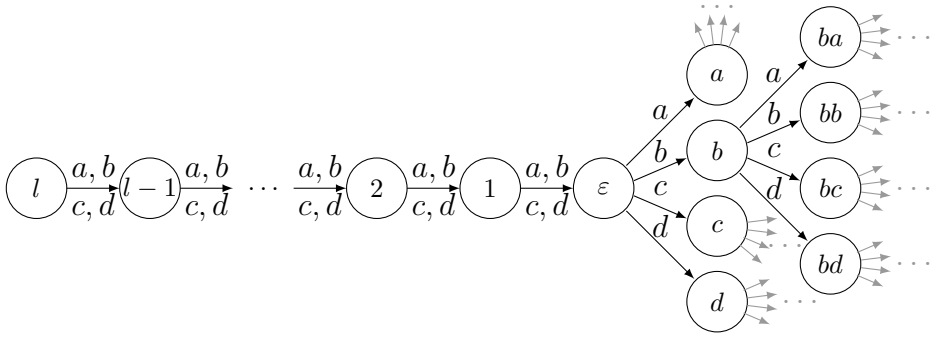

FIGURE 6.1 – Système de transition de l'objet l-countdown-append. L'opération de lecture non représentée renvoie le label de l'état sur lequel elle est appliquée.

Objet Countdown-append L'objet l-countdown-append, où $l \in \mathbb{N}$, comporte 4 opérations de mise à jour, $U = \{a, b, c, d\}$ et une opération de requête, q. La figure 6.1 représente le comportement de l'objet en tant que système de transition d'état. Il est divisé en deux étapes : lors de la première, l'objet décompte le nombre d'opérations d'écriture, en partant de l, jusqu'à 1, puis ε (le mot vide). Dans la deuxième étape, les opérations d'écriture sont concaténées à la fin de l'état. Enfin, l'opération de requête renvoie l'état local des objets à chaque exécution.

Pour toutes les séquences $v = u_1...u_l \in U^l$ comprenant l opérations de mise à jour de l'objet l-countdown-append, nous désignons par H_v l'histoire partagée dans laquelle un processus effectue toutes les mises à jour de v dans leur ordre d'apparition dans v. Dans les sections suivantes, nous étudions la H_v-complexité des algorithmes dans le modèle opérationnel et dans le modèle *wait-free*.

Borne inférieure dans le modèle opérationnel Nous prouvons maintenant que pour tout algorithme du modèle opérationnel, il existe un certain v tel que l'algorithme a une H_v-complexité d'au moins $\frac{l}{2} - 1$ bits. Notre preuve suit le schéma introduit dans [Bur+14] : nous construisons une famille d'exécutions de telle sorte qu'à un moment donné de l'exécution, le processus p_i effectuant les opérations de v est incapable de faire la distinction entre toutes ces exécutions et une exécution modélisée par H_v. Ensuite, à un stade ultérieur de l'exécution, p_i doit être capable de distinguer suffisamment d'exécutions différentes afin de maintenir la convergence possible.

Théorème 7. *Pour tout algorithme déterministe A qui implémente un objet l -countdown-append update cohérent dans le modèle opérationnel, il existe v tel*

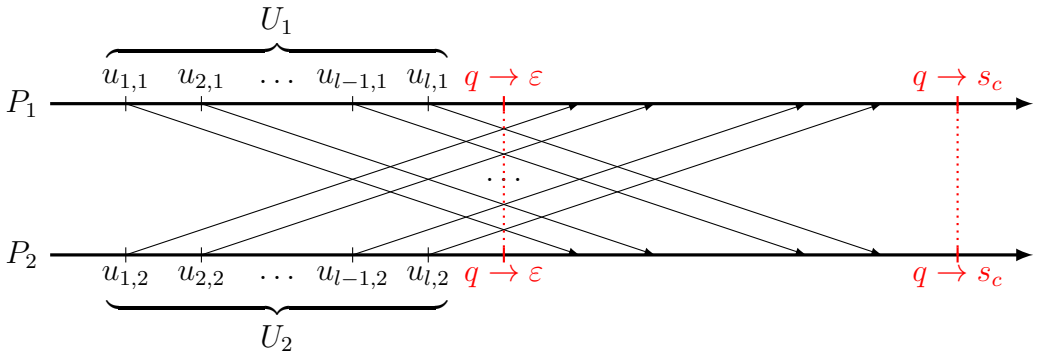

FIGURE 6.2 – Exécution typique utilisée dans la preuve du théorème 7

que la H_v-complexité de A est au moins $\frac{l}{2} - 1$ bits.

Démonstration. Soit A un algorithme du modèle opérationnel implémentant un objet l-countdown-append *update* cohérent. Pour chaque paire de séquences d'opérations de mise à jour (v_1, v_2), où $v_1 \in \{a, b\}^l$ et $v_2 \in \{c, d\}^l$, nous définissons l'exécution $X_{(v_1,v_2)}$, illustrée sur la figure 6.2, comme suit. Seuls deux processus p_1 et p_2 effectuent des étapes dans $X_{(v_1,v_2)}$. Tous les autres processus tombent en panne avant le début de l'exécution. Initialement, le processus p_1 (resp. p_2) exécute séquentiellement, dans l'ordre, les opérations formant v_1 (resp. v_2). Conformément au modèle opérationnel, ils diffusent un seul message lors de chaque opération. Dans un second temps, ils reçoivent tous les deux les messages des autres, en respectant l'ordre FIFO (rappelons que la diffusion causale, définie page 39, implique un ordre FIFO). Enfin, les deux processus effectuent une opération de requête. Nous désignons par $\mathcal{X} = \{X_{(v_1,v_2)} : v_1 \in \{a, b\}^l \wedge v_2 \in \{c, d\}^l\}$ l'ensemble de toutes les exécutions $X_{(v_1,v_2)}$.

Remarquons tout d'abord que la cohérence d'écriture impose que les deux opérations de requête retournent la même valeur v_c, qui est un suffixe de taille l, d'un entrelacement de v_1 et v_2. Soit $f(v_1, v_2)$ le nombre d'opérations c et d dans v_c. Notez que f est bien défini car A est déterministe.

Les exécutions peuvent être distinguées en fonction du processus qui a la majorité des opérations dans l'état de convergence. Nous définissons $\mathcal{X}_1 = \{X_{(v_1,v_2)} \in \mathcal{X} : f(v_1, v_2) \geq \frac{l}{2}\}$ et $\mathcal{X}_2 = \mathcal{X} \setminus \mathcal{X}_1$. Comme $\mathcal{X}_1$ et $\mathcal{X}_2$ forment une partition de $\mathcal{X}$ qui a une taille 2^{2l}, nous avons $|\mathcal{X}_1| \geq 2^{2l-1}$ ou $|\mathcal{X}_2| \geq 2^{2l-1}$. Sans perte de généralité, nous supposons que $|\mathcal{X}_1| \geq 2^{2l-1}$.

Nous partitionnons maintenant $\mathcal{X}_1$ en fonction de la valeur de v_1. Pour chaque

séquence $v_1 \in \{a,b\}^l$, soit $\mathcal{X}_1(v_1) = \{X_{(v,v_2)} \in \mathcal{X}_1 : v = v_1\}$. Comme il y a 2^l valeurs possibles pour v_1, au moins l'une d'entre elles est telle que $|\mathcal{X}_1(v_1)| \geq \frac{|\mathcal{X}_1|}{|\{a,b\}^l|} = \frac{2^{2l-1}}{2^l} = 2^{l-1}$. Fixons un tel v_1.

Soient v_2 et v_2' telles que $X_{(v_1,v_2)}$ et $X_{(v_1,v_2')}$ appartiennent à $\mathcal{X}_1(v_1)$. Par définition de f, si $X_{(v_1,v_2)}$ et $X_{(v_1,v_2')}$ convergent vers le même état, alors v_2 et v_2' diffèrent au plus par leurs $l - f(v_1,v_2) \leq \frac{l}{2}$ premières opérations. Par conséquent, il y a au moins $\frac{2^{l-1}}{2^{\frac{l}{2}}} = 2^{\frac{l}{2}-1}$ valeurs différentes de v_2 pour lesquelles $X_{(v_1,v_2)}$ conduisent à des états de convergence différents. Soit $\mathcal{X}'$ un sous-ensemble de $\mathcal{X}_1(v_1)$ de taille $2^{\frac{l}{2}-1}$, dans lequel tous les états de convergence sont différents.

Dans le modèle opérationnel, l'état local du processus p_2 à la fin de l'exécution ne dépend que de son état local après avoir exécuté ses l propres opérations de mise à jour, et des messages reçus de p_1 par la suite. Dans toutes les exécutions de $\mathcal{X}'$, les messages reçus par p_2 sont les mêmes dans toutes les exécutions car v_1 est fixé. De plus, l'état local de p_2 à la fin de toutes les exécutions est différent. Cela signifie que l'état local de p_2 après avoir exécuté ses opérations de mise à jour est également différent dans toutes les exécutions. Par conséquent, il y a une séquence v_2 telle que, après avoir exécuté toutes les opérations de mise à jour dans v_2 (exécution X), l'état local de p_2 nécessite au moins $\frac{l}{2} - 1$ bits.

Enfin, considérons l'exécution X dans laquelle seulement p_2 effectue des pas de calcul, exécutant la séquence des opérations de mise à jour de v_2. Juste après avoir exécuté ses mises à jour, p_2 ne peut pas faire la distinction entre les exécutions X et $X_{v_1}^{v_2}$, donc son état local dans X nécessite également $\frac{l}{2} - 1$ bits. De plus, X est modélisé par H_{v_2}. Par conséquent, la H_{v_2}-complexité de A est d'au moins $\frac{l}{2} - 1$ bits.

□

Borne supérieure dans le modèle *wait-free* Une conséquence du théorème 7 est que toute construction universelle update cohérente dans le modèle opérationnel doit conserver des métadonnées dont la taille est au moins proportionnel au nombre d'opérations de mise à jour dans certaines exécutions. Bien que cela n'implique pas que ces métadonnées doivent être stockées sous la forme d'un historique contenant toutes les opérations de mise à jour comme dans l'algorithme 7, cela implique que les métadonnées requises par UC_∞ ne peuvent pas être considérablement réduites, et en particulier qu'il est impossible de sup-

```
1  Opération apply(o ∈ O) ∈ R
2      var s ∈ S ← state_i;
3      if o is an update then broadcast mUpdate(o) ;
4      return q(s, o);
5  Quand un message mUpdate (o_j ∈ O) est reçu depuis p_j
6      state_i ← u(state_i, o_j) ; clock_i[j] ← clock_i[j] + 1 ; leader_i ← i;
7      broadcast mCorrect (clock_i, state_i);
8  Quand un message mCorrect (cl_j ∈ ℕ^ℕ, s_j ∈ S) est reçu depuis p_j
9      if clock_i = cl_j ∧ j ≤ leader_i then
10         state_i ← s_j ; leader_i ← j;
```

Algorithme 8 : Construction universelle $UC_0(\mathcal{O}, \mathcal{R}, \mathcal{S}, s_0, u, q)$: code pour p_i

primer en toute sécurité des anciennes opérations de l'histoire de l'algorithme 7 dans le modèle opérationnel. Différemment, le théorème 7 ne s'applique pas dans le modèle plus général *wait-free*. Dans cette section, nous proposons l'algorithme 8, appelé UC_0, dont les métadonnées consistent principalement en une horloge vectorielle de taille $O(n \log(m))$ où n est le nombre de processus et m le nombre total d'opérations de requête émises.

Le principe de UC_0 est de dissocier les parties sécurité et vivacité de la cohérence d'écriture. Du côté de la sécurité, chaque processus maintient sa propre vision de l'état de l'objet, en s'assurant qu'il résulte d'une éventuelle linéarisation de toutes les opérations de mise à jour dont il a connaissance. Cela garantit que, si tous les processus sauf un tombent en panne, toutes les opérations de requête effectuées par le processus restant sont effectuées dans un état cohérent. Du côté de la vivacité, les processus échangent leurs visions de l'etat, s'efforçant de converger vers un état commun. Plus précisément, pour un ensemble donné d'opérations de mise à jour, représenté par un vecteur de versions, les processus adopteront toujours l'état proposé par le processus avec le plus petit identifiant, qu'ils appellent leur *leader*. Cela garantit la cohérence d'écriture puisque, si tous les processus cessent d'effectuer des opérations de mise à jour, tous les processus corrects finiront par adopter l'état façonné par le processus correct avec le plus petit identifiant.

Le processus p_i maintient trois variables. La variable $\mathsf{state}_i \in \mathcal{S}$ représente l'état local courant de p_i, initialisé avec l'état initial s_0. La variable $\mathsf{clock}_i \in \mathbb{N}^{\mathbb{N}}$

est l'équivalent d'un vecteur de version, telle que $\mathsf{clock}_i[j]$ représente le nombre d'opérations de mise à jour reçues par p_i du processus p_j. Comme p_i ne connaît pas le nombre de participants, il est encodé sous forme de tableau associatif, plutôt qu'un vecteur. Enfin, la variable $\mathsf{leader}_i \in \mathbb{N}$ est l'identifiant d'un processus telle que, si $\mathsf{clock}_i = \mathsf{clock}_{\mathsf{leader}_i}$, alors p_i et p_{leader_i} sont dans le même état local.

Lorsque le processus p_i appelle une opération de requête, il l'exécute localement sur son état local state_i (Ligne 4). Lorsque p_i appelle une opération de mise à jour, il diffuse un message `mUpdate` (Ligne 3). A la réception d'un tel message de p_j, p_i exécute l'opération sur son état local et met à jour son horloge vectorielle. Il réinitialise également sa variable leader_i à i, ce qui signifie que son état local a été calculé par lui-même. Ensuite, p_i diffuse un message `mCorrect` contenant sa version actuelle de l'état et le vecteur de version associé. Lorsque p_i reçoit un message `mCorrect` de p_j, il vérifie d'abord la version de l'état en comparant la version cl_j du message avec son propre vecteur de version clock_i. Grâce à la diffusion causale, clock_i ne peut pas être plus ancien que cl_j. Si $cl_j[k] < \mathsf{clock}_i[k]$ pour un k, les dernières mises à jour de p_k sont déjà connues par p_i, mais pas par p_j. Dans ce cas, le processus p_i ignore simplement le message. Sinon, si $cl_j = \mathsf{clock}_i$, p_i conserve l'état calculé par le processus avec le plus petit identifiant, dont il se souvient en mettant à jour sa variable leader_i.

Théorème 8 (Exactitude)**.** *L'algorithme 8 est une construction universelle wait-free, update cohérente.*

Démonstration. L'opération `apply` dans l'algorithme 8 se termine car elle ne contient pas de boucle.

Soit H un historique admis par l'algorithme 8. Si H contient un nombre fini de requêtes ou un nombre infini de mises à jour, elle est *update* cohérente par définition. Supposons que H contienne un nombre infini de requêtes et un nombre fini m de mises à jour (nous ne considérons que les mises à jour pour lesquelles un message `mUpdate` a été envoyé). Notons m_i le nombre de mises à jour effectuées par p_i, tel que $m = \sum_{i=1}^{n} m_i$.

Certains processus sont corrects car au moins l'un d'entre eux effectue un nombre infini de requêtes, et tous les processus corrects p_i envoient un message $\texttt{mCorrect}(cl_{max}, s_i)$, avec $cl_{max} = [m_1, \ldots, m_n]$, après avoir reçu tous les messages `mUpdate` (Ligne 7). Soit p_{leader} le processus avec le plus petit identifiant qui a envoyé un tel message, appelé m_{leader}. Grâce à la diffusion causale, tout processus

correct p_i a reçu tous les messages `mUpdate` avant m_{leader}, donc à la réception de m_{leader}, il avait $\mathsf{clock}_i = cl_{max}$ et, par définition de p_{leader}, $leader < \mathsf{leader}_i$. Par conséquent, p_i a adopté s_{leader} comme son propre état, et pour toute réception ultérieure d'un message $\texttt{mCorrect}(cl_j, s_j)$ par p_i, soit $cl_j < \mathsf{clock}_i = cl_{max}$, soit $leader = \mathsf{leader}_i < j$. Par conséquent, tous les processus corrects p_i convergent vers le même état s_{leader}.

Prouvons maintenant qu'à tout instant $\mathsf{clock}_i[j]$ représente le nombre de `mUpdate` reçu par p_i de p_j, et que l'état state_i de p_i peut être obtenu par une linéarisation des opérations de mise à jour correspondantes. Nous le prouvons par récurrence sur le nombre de fois que state_i et clock_i ont changé pour un certain processus p_i. Initialement, $\mathsf{clock}_i[j] = 0$ et $\mathsf{state}_i = s_0$. Supposons que la propriété soit toujours vraie jusqu'à ce que le processus p_i mette à jour $\mathsf{clock}_i[j]$ ou state_i. Si cela se produit après la réception d'un message `mUpdate` de p_j, la propriété reste vraie car la diffusion causale implique une réception FIFO. Si cela se produit après la réception d'un message `mCorrect` de p_j, alors la propriété était auparavant vraie sur le processus p_j, sur l'horloge cl_j et sur l'état s_j, elle reste donc vraie après la mise à jour de l'horloge $\mathsf{clock}_i = cl_j$ et de l'état $\mathsf{state}_i = s_j$.

Enfin, toutes les opérations de requête effectuées après la réception de tous les messages sont exécutées dans le même état s_{leader}, qui peut être obtenu par une linéarisation de toutes les opérations de mise à jour. Cela implique la cohérence d'écriture. □

Nous pouvons enfin conclure sur la non-équivalence entre les deux modèles de calcul pour la mise en œuvre de la cohérence d'écriture.

Corollaire 2. *Il existe un objet O et un algorithme A_{wf} implémentant un objet update cohérent O dans le modèle wait-free, tels que, pour tout algorithme A_{om} implémentant un objet update cohérent O dans le modèle opérationnel, il existe un historique H tel que A_{wf} a une complexité H strictement inférieure à celle de A_{om}.*

Démonstration. Soit $l \in \mathbb{N}$ et soit O_l l'objet l-countdown-append. Soit $v \in U^l$. Dans toute exécution de l'algorithme 8 abstrait par H_v, il existe un processus p_i qui effectue toutes les l opérations de mise à jour. A la fin de l'exécution, clock_j ne contient qu'une seule paire (i, l) qui peut être encodé en $O(\log(nl))$ bits, $\mathsf{state}_j = \varepsilon$ a une taille constante, et leader_j est l'identifiant d'un processus,

de taille $O(\log(n))$ bits. Par conséquent, il existe une constante $x > 1$ telle que, pour tout v, la complexité H_v de l'algorithme 8 est strictement inférieur à $x \log(nl)$ bits.

Soit $l > 2^{-\frac{1+x \log n}{x}}$. Nous avons $x \log(nl) < \frac{l}{2} - 1$. Soit O l'objet l-countdown-append, et soit A_{om} une implémentation *update* cohérente de O dans le modèle opérationnel. Par le théorème 7, il existe v tel que la complexité H_v de A_{om} est d'au moins $\frac{l}{2} - 1$ bits. Par conséquent, l'algorithme 8 a une complexité H_v strictement inférieure à celle de A_{om}. □

6.3 Une solution pratique

Dans les sections précédentes, nous avons rencontré deux stratégies différentes afin de mettre en œuvre une construction universelle *update* cohérente. D'une part, l'algorithme UC_∞ appartient au modèle opérationnel, et a donc une complexité optimale en nombre de messages échangés. La limite de cet algorithme est qu'il nécessite une quantité illimitée de mémoire et de puissance de calcul, car l'historique stocké peut croître indéfiniment. Selon le théorème 7, cette limite ne peut être outrepassée tout en restant dans le modèle opérationnel. D'autre part, l'algorithme UC_0 parvient à se débarrasser du stockage de l'historique grâce à un mécanisme de synchronisation interdit dans le modèle opérationnel. La limitation de cet algorithme est sa complexité en terme de communication réseau, car un état complet est diffusé par tous les processus pour toutes les mises à jour.

Cette section présente l'algorithme $UC[k]$, qui utilise le meilleur des deux méthodes. Il s'appuie sur le constat que, dans les systèmes réels, l'asynchronie est souvent utilisée comme une abstraction commode pour les systèmes dans lesquels les délais de transmission sont en fait bornés, mais la borne est trop grande pour être utilisée en pratique, ou inconnue. Cela signifie qu'après un certain temps, les anciens messages pour la première stratégie peuvent être supprimés (*garbage collection*). La deuxième stratégie est utilisée pour assurer la sécurité de ce nettoyage : si les anciens messages sont reçus plus tard que prévu, l'ordre total *a priori* peut être modifié et les états doivent être échangés pour assurer la convergence. Ainsi, le surcoût en mémoire et en temps de calcul reste limité par un paramètre k qui affecte la taille de la liste, alors qu'une bande passante plus importante n'est requise que lorsque les messages sont anormalement retardé, par

exemple pour se remettre d'une partition. UC_∞ est la limite de $UC[k]$ quand $k \to \infty$, et $UC[0]$ correspond à une amélioration de UC_0. La valeur k peut être vue comme un paramètre qui combine le taux d'émission des opérations de mise à jour et le délai de transmission du message.

```
1  Opération apply(o ∈ O) ∈ R
2      var s ∈ S ← state_i;
3      if o est une requête then
4          for (t, j, o') ∈ history_i triés selon (t, j) do s ← u(s, o') ;
5      if o est une mise à jour then time_i ← time_i + 1 ; broadcast
         mUpdate (time_i, o) ;
6      return q(s, o);
7  record(t ∈ ℕ)
8      rtime_i ← max(rtime_i, t);
9      for (t_j, j, o_j) ∈ history_i avec t_j ≤ rtime_i triés selon (t_j, j) do
10         state_i ← u(state_i, o_j) ; history_i ← history_i \ {(t_j, j, o_j)};
11         clock_i[j] ← clock_i[j] + 1 ; leader_i ← i ; sent_i ← false ;
12 Quand un message mUpdate(t_j ∈ ℕ, o_j ∈ O) est reçu depuis p_j
13     time_i ← max(time_i, t_j);
14     history_i ← history_i ∪ {(t_j, j, o_j)};
15     var conflict ∈ {true, false} ← t_j ≤ rtime_i;
16     record(time_i − k);
17     if conflict then sent_i ← true ; broadcast
         mCorrect(clock_i, rtime_i, state_i) ;
18 Quand un message mCorrect(cl_j ∈ ℕ^ℕ, t_j ∈ ℕ, s_j ∈ S) est reçu
     depuis p_j
19     record(t_j);
20     if clock_i = cl_j ∧ j < leader_i then
21         state_i ← s_j ; leader_i ← j ; sent_i ← true ;
22     else if ¬sent_i then
23         sent_i ← true ; broadcast mCorrect (clock_i, rtime_i, state_i);
```

Algorithme 9 : Construction universelle $UC[k](\mathcal{O}, \mathcal{R}, \mathcal{S}, s_0, u, q)$: code pour p_i

Présentation de la construction universelle Le code pour $UC[k]$ exécuté par le processus p_i est donné sur l'algorithme 9. L'état local actuel de l'objet au processus p_i est divisé en deux parties. D'une part, les opérations de mise à jour les plus récentes connues par p_i sont stockées avec des horodatages dans une liste

de mises à jour, de la même manière que dans l'algorithme UC_∞. D'autre part, les anciennes mises à jour sont supprimées et enregistrées dans un état de l'objet, maintenu de la même manière que dans l'algorithme UC_0. La séparation entre les opérations de mise à jour anciennes et récentes est arbitraire et abstraite par le paramètre k de l'algorithme : à tout moment, la différence entre tous les horodatages de la liste des mises à jour récentes doit être au plus de k. Contrairement à l'algorithme UC_0, l'état local de l'objet n'est pas envoyé à chaque fois qu'une opération de mise à jour est reçue, mais seulement après qu'un conflit ait été détecté : un conflit survient chez p_i lorsque p_i reçoit un message `mUpdate` pour une opération de mise à jour, dont l'horodatage indique qu'il devrait déjà faire partie de l'état enregistré. La contrepartie de cette optimisation est qu'un message `mCorrect` peut être envoyé par p_i en réponse à un autre message `mCorrect` envoyé par p_j, si p_j a détecté un conflit mais que p_i ne l'a pas fait. Chaque processus p_i gère les sept variables locales suivantes. Les variables $\mathsf{time}_i \in \mathbb{N}$, initialisée à 0, et $\mathsf{history}_i \subset \mathbb{N} \times \mathbb{N} \times \mathcal{O}$, initialement vide, sont respectivement l'état de l'horloge de Lamport et la liste ordonnée des opérations de mise à jour récentes. Elles jouent un rôle similaire à celui qu'elles ont dans l'algorithme UC_∞, excepté que $\mathsf{history}_i$ est régulièrement « nettoyé » de ses plus anciennes mises à jour. Les variables $\mathsf{state}_i \in \mathcal{S}$, initialisée à s_0, $\mathsf{clock}_i \in \mathbb{N}^{\mathbb{N}}$, initialisée à $[0, \ldots, 0]$, et $\mathsf{leader}_i \in \mathbb{N}$, initialement à i, jouent le même rôle que dans l'algorithme UC_0, mais n'encodent que les anciennes mises à jour : state_i est l'état enregistré calculé à partir d'anciennes mises à jour, et clock_i est le vecteur de version qui décrit l'ensemble des opérations de mise à jour utilisées par p_{leader_i} pour créer state_i. La variable $\mathsf{rtime}_i \in \mathbb{N}$, initialisée à $-k$, encode l'horodatage qui sépare les anciennes opérations de mise à jour des récentes. Normalement, $\mathsf{rtime}_i = \mathsf{time}_i - k$, mais elle peut être différente si les processus ne s'accordent pas sur la valeur de k, comme indiqué dans la section 23. Enfin, sent_i est un booléen, initialisé à **true**, qui code si state_i a été diffusé ou non sur le réseau, afin de minimiser le trafic.

Lorsque le processus p_i exécute une opération de requête, il applique d'abord toutes les opérations de mise à jour stockées dans $\mathsf{history}_i$ sur state_i selon l'ordre lexicographique sur leurs horodatages et calcule la valeur de retour en appliquant la requête à l'état obtenu.

Lorsque le processus p_i exécute une opération de mise à jour o, il diffuse le message $\texttt{mUpdate}(t, o)$, où o identifie l'opération de mise à jour, et t est le temps

virtuel généré à l'aide de l'horloge de Lamport time_i utilisé dans l'horodatage de l'opération.

Lorsque le processus p_i reçoit un message $\texttt{mUpdate}(t_j, o_j)$ de p_j, il met d'abord à jour son horloge virtuelle locale time_i et insère o_j dans la liste de mise à jour $\mathsf{history}_i$. Ensuite, p_i applique toutes les opérations de mise à jour dont les horodatages sont inférieurs à $\mathsf{time}_i - k$ à state_i en exécutant $\texttt{record}(\mathsf{time}_i - k)$. Deux cas sont possibles, selon les valeurs relatives de t_j et de rtime_i à la réception du message. Normalement, $\mathsf{rtime}_i < t_j$, ce qui signifie que o_j est une opération de mise à jour récente qui appartient à $\mathsf{history}_i$, et p_i n'a rien d'autre à faire. Sinon, o_j devrait déjà avoir été inclus dans state_i. En exécutant o_j sur state_i, p_i peut avoir compromis la convergence puisqu'un autre processus peut avoir exécuté les opérations dans un ordre différent. En appliquant l'opération de mise à jour à state_i, p_i a changé l'ordre de linéarisation, mais ce nouvel ordre est correct grâce à la réception causale. Il notifie donc les autres processus de son choix en diffusant un message `mCorrect` contenant son nouvel état.

Lorsque le processus p_i reçoit un message $\texttt{mCorrect}(cl_j, t_j, s_j)$ de p_j, il sait que p_j a fait face à un conflit et a changé son ordre de linéarisation. Il peut accepter ou rejeter la correction, selon le vecteur de version cl_j associé à l'état s_j de la correction, et au vecteur de version clock_i associé à son propre état state_i. Grâce à la diffusion causale, p_i a reçu au moins les mêmes messages `mUpdate` avant le message `mCorrect` que p_j a reçu avant l'envoi du message `mCorrect`. Le processus p_i exécute ensuite $\texttt{record}(t_j)$ pour s'assurer que $\mathsf{clock}_i \geq cl_j$ (cela n'est pas nécessaire si tous les processus partagent la même valeur pour k. Les deux cas suivants sont possibles :

— Si $\mathsf{clock}_i = cl_j$, les deux états ont été produits avec les mêmes mises à jour mais peut-être dans un ordre différent. Dans ce cas, l'arbitrage est effectué en considérant le processus qui a le plus petit identifiant. Si $j < \mathsf{leader}_i$ alors p_i accepte la correction et met à jour ses variables locales. Sinon, p_i choisit de garder son propre état. Il le diffuse pour que chacun puisse également adopter son état. Si cette situation se produit plusieurs fois de suite, il est nécessaire de comparer tous les identifiants entre eux, donc la comparaison se fait avec la variable leader_i, qui contient l'identifiant du processus dont l'état a été choisi. La variable sent_i est utilisée pour empêcher p_i de diffuser plusieurs fois le même état.

— Si $\mathsf{clock}_i > cl_j$ alors state_i prend en compte les mises à jour, ce que s_j ne fait pas. La réception future de ces mises à jour par p_j entraînera un nouveau conflit, donc p_i envoie un autre message `mCorrect` pour aider p_j à résoudre le conflit.

Exactitude Dans cette section, nous prouvons que l'algorithme 9 implémente la cohérence d'écriture, c'est-à-dire que tous les historiques qu'il permet sont *update* cohérents. À cette fin, nous prouvons trois lemmes intermédiaires : Le lemme 21 assure, parmi d'autres propriétés concernant l'état global du système, que l'état local virtuel d'un processus est toujours valide par rapport aux opérations de mise à jour dont il a entendu parler. Le lemme 22 prouve que, si tous les processus arrêtent d'effectuer des mise à jour, alors au bout d'un certain temps, plus aucun message ne sera envoyé pour maintenir la cohérence et dans cette situation, le lemme 23 prouve que, tous les processus ont convergé vers un état commun. Enfin, le théorème 9 prouve que l'algorithme 9 est *update* cohérent. Remarquez qu'à chaque fois qu'une opération est émise, elle se termine sans attendre aucun autre processus, donc l'algorithme est *wait-free*.

On considère un objet $O = (\mathcal{O}, \mathcal{R}, \mathcal{S}, s_0, u, q)$ et un historique H qui modélise une exécution de l'algorithme 9. Introduisons les notations suivantes.

— La notation en exposant sur les variables de l'algorithme, par ex. state_i^x, désigne la valeur d'une variable à l'instant x.

— $vs_i^x \in \mathcal{S}$ désigne l'état virtuel du processus p_i à l'instant x, obtenu en exécutant les opérations contenues dans $\mathsf{history}_i^x$ sur state_i^x, en respectant l'ordre lexicographique sur leurs horodatages, de manière similaire à laquelle les requêtes sont faites sur, puis contenues dans la variable s après la ligne 4.

— $vcl_i^x \in \mathbb{N}^{\mathbb{N}}$ désigne l'horloge virtuelle du processus p_i à l'instant x, qui représente l'ensemble des messages `mUpdate` reçus par p_i à l'instant x. Elle est définie, pour tout $j \in \{1, \ldots, n\}$, par $vcl_i^x[j] = \mathsf{clock}_i^x[j] + |\{(t, k, o) \in history_i^x : k = j\}|$.

— Pour $cl \in \mathbb{N}^{\mathbb{N}}$ et $s \in \mathcal{S}$, $cl \triangleright s$ exprime le fait que le préfixe de l'historique contenant les $cl[j]$ premières opérations de mise à jour de chaque processus p_j peut conduire à l'état s.

Lemme 21 (Validité)**.** *A chaque instant x, pour chaque processus p_i qui n'exécute*

pas une partie de l'algorithme, et pour chaque message $\texttt{mCorrect}(cl_i, t_i, s_i)$ *envoyé par* p_i *avant l'instant* x*, nous avons :*

1. $vcl_i^x \triangleright vs_i^t$,
2. $\mathsf{clock}_i^x \triangleright \mathsf{state}_i^x$,
3. $cl_i \triangleright s_i$.

Démonstration. Nous procédons par induction sur la succession des états globaux du système. Au départ, il n'y a pas de message en transit, et pour chaque processus p_i, $[0, ..., 0] = \mathsf{clock}_i^0 = vcl_i^0 \triangleright vs_i^0 = \mathsf{state}_i^0 = s_0$. Nous supposons maintenant qu'à un instant x^-, la propriété est vérifiée et l'état global change. Nous prouvons que la propriété est toujours vérifiée à x^+, juste après les changements, que nous considérons se produire sur p_i. Les variables mentionnées dans le lemme ne sont modifiées qu'à la réception d'un message, il faut donc distinguer deux cas.

Réception de $\texttt{mUpdate}(t_j, o_j)$ **envoyé par** p_j**.** Avant d'appeler la fonction `record`, $vcl_i^x[j]$ a été incrémenté et vs_i^x a été mis à jour par l'insertion de o_j dans $\mathsf{history}_i$. Comme la diffusion causale assure la réception FIFO des messages de p_j par p_i, (1) tient toujours, tandis que (2) et (3) restent inchangés.

A chaque itération de la boucle de `record`, ni vcl_i^x ni vs_i^x ne sont modifiés, car l'ordre dans lequel les mises à jour sont triées sur la ligne 9 et dans la définition de vs_i^x sont les mêmes, donc (1) reste valide. Après la boucle, (2) est toujours vraie car les opérations de mise à jour sont appliquées et supprimées de $\mathsf{history}_i$ dans un ordre défini par une horloge de Lamport, qui contient l'ordre de processus.

Les messages `mCorrect` présents dans le système sont ceux présents à t^-, qui vérifie (3), et éventuellement celui envoyé à la ligne 17, avec $cl_i = \mathsf{clock}_i^{x+} \triangleright \mathsf{state}_i^{x+} = s_i$ donc (3) reste valide.

Réception de $\texttt{mCorrect}(cl_j, t_j, s_j)$ **envoyé par** p_j**.** Comme ci-dessus, les propriétés sont respectées après l'appel à `record` (instant x_R).

Si la condition de la ligne 20 est vraie, (3) reste valide car aucun nouveau message $\texttt{mCorrect}(cl_i, t_i, s_i)$ n'est envoyé par p_i et (2) tient car $\mathsf{clock}_i^{x^+} = cl_j \triangleright s_j = \mathsf{state}_i^{x^+}$. L'horloge virtuelle $vcl_i^{x^+}$ reste inchangée et la concaténation de la linéarisation donnée par (2) et de toutes les opérations de mise

à jour de $\mathsf{history}_i^{x^+}$ est toujours une linéarisation valide qui conduit à $vs_i^{x^+}$ grâce à la diffusion causale, donc (1) reste valide.

Si la condition est fausse, (1) et (2) ne sont pas impactées, et (3) reste valide car le message $\texttt{mCorrect}(cl_i, t_i, s_i)$ envoyé par p_i est tel que $cl_i = \mathsf{clock}_i^{x^+} \triangleright \mathsf{state}_i^{x^+} = s_i$.

□

Lemme 22 (Silence)**.** *Si chaque processus p_i effectue un nombre fini m_i d'opérations de mise à jour, il existe un instant x_{quiet} tel que, pour tout $x > x_{\text{quiet}}$, aucun message n'est en transit dans le réseau à l'instant x, et pour tous les processus p_i, $vcl_i^x = [m_1, ..., m_n]$.*

Démonstration. Comme les messages `mUpdate` ne sont diffusés qu'à la ligne 5, lorsqu'une nouvelle mise à jour est effectuée, seul un nombre fini d'entre eux sont envoyés. De plus, p_i ne peut pas envoyer deux messages `mCorrect` avec la même valeur de clock_i : si l'envoi se fait en ligne 17, *conflict* est vrai et donc clock_i a été incrémenté juste avant ou lors de l'appel de `record` (clock_i est incrémentée car au moins l'opération de mise à jour reçue est appliquée car il y a eu un conflit), et si la diffusion se fait sur la ligne 23, sent_i a été mis à faux dans la fonction `record` juste après un incrément de clock_i. Comme clock_i est bornée par $[m_1, ..., m_n]$, un nombre limité de messages `mCorrect` est envoyé. Comme tous les messages arrivent finalement, il y a un instant x_{quiet} tel que pour tout $x > x_{\text{quiet}}$, tous les messages ont été reçus et traités. En particulier, tous les messages `mUpdate` ont été reçus par p_i, donc $vcl_i^{x_{\text{quiet}}} = [m_1, .., m_n]$. □

Lemme 23 (Convergence)**.** *Pour tout instant x, s'il n'y a pas de message en transit à x, alors pour tous les processus p_i et p_j, $vs_i^x = vs_j^x$.*

Démonstration. Soit x un instant auquel aucun message n'est en transit, et soient p_i et p_j deux processus.

Supposons d'abord qu'un conflit se soit produit lors de l'exécution, et qu'un message `mCorrect` ait été envoyé. Considérons un message m_k de la forme $\texttt{mCorrect}(cl_k, t_k, s_k)$ envoyé par p_k tel qu'il n'y a pas d'autre message $\texttt{mCorrect}(cl_l, t_l, s_l)$ envoyé par p_l avec $cl_l > cl_k$, ou $cl_l = cl_k$ et $l < k$ (m_k peut ne pas être unique car l'ordre sur les vecteurs de version est partiel, mais il existe).

Comme aucun message n'est en transit à l'instant x, p_i a déjà reçu m_k, et après l'exécution de `record` à l'instant x_i, nous avons $\mathsf{clock}_i^{x_i} \geq cl_k$ grâce à la

diffusion causale. Nous savons que $\mathsf{clock}_i^{x_i} = cl_k$ et $k < \mathsf{leader}_i^{x_i}$ puisque, sinon, $p_{\mathsf{leader}_i^{x_i}}$ aurait envoyé un message $m_i = \texttt{mCorrect}(\mathsf{clock}_i^{x_i}, t, \mathsf{state}_i^{x_i})$, soit parce que p_i l'a reçu si $\mathsf{leader}_i^{x_i} \neq i$, ou parce que p_i a exécuté la ligne 23 si $\mathsf{sent}_i^{x_i}$ était faux ou la ligne 23 si $\mathsf{sent}_i^{x_i}$ était vrai. Dans tous les cas, m_i contredit la définition de m_k donc il ne peut pas exister. Par conséquent, p_i a exécuté la ligne 21, et a accepté la correction. De plus, aucun autre conflit n'est survenu chez p_i car un tel conflit se traduirait par un message `mCorrect` associé à un vecteur de version strictement supérieur à cl_k, contredisant la définition de m_k. De même, p_j a adopté la correction de m_k à un instant donné x_j et n'a fait l'objet d'aucun conflit par la suite.

Si aucun conflit n'est survenu lors de l'exécution, posons $x_i = x_j = 0$ l'instant initial. Dans les deux cas, nous avons $\mathsf{clock}_i^{x_i} = \mathsf{clock}_j^{x_j}$, $\mathsf{state}_i^{x_i} = \mathsf{state}_j^{x_j}$, et aucun conflit n'est survenu chez p_i après x_i, ni chez p_j après x_j. Ensuite, chaque message $\texttt{mUpdate}(t_k, o_k)$ reçu par p_i à un instant $x_r > x_i$ est associé à un temps virtuel $t_k > \mathsf{rtime}_i^{x_r}$, et la même chose peut être dite à propos de p_j. Par conséquent, p_i et p_j appliquent toutes les opérations de mise à jour dans `record` selon le même ordre lexicographique, ce qui les conduit au même état virtuel $vs_i^x = vs_j^x$. □

Théorème 9 (Exactitude)**.** *Toutes les histoires autorisées par l'algorithme 9 sont update cohérentes.*

Démonstration. Soit H un historique autorisé par l'algorithme 9. Si H contient un nombre infini de mises à jour ou un nombre fini de requêtes, il est *update* cohérent par définition. Supposons que chaque processus p_i effectue un nombre fini m_i d'opérations de mise à jour. Selon le lemme 22, il existe un instant x_{quiet} tel que, pour tout $x > x_{\text{quiet}}$, aucun message n'est en transit dans le réseau à x, et pour tout i, $vcl_i^x = [m_1, ..., m_n]$. Par le lemme 23, il y a un état $s_{\text{conv}} \in \mathcal{S}$ tel que, pour tout processus p_i et tout $x > x_{\text{quiet}}$, $vs_i^x = s_{\text{conv}}$. De plus, comme $vcl_i^x = [m_1, ..., m_n]$ et selon le lemme 21.1, $vcl_i^x \triangleright vs_i^x$, ce qui signifie qu'il y a une linéarisation de toutes les opérations de mise à jour qui conduisent à cet état commun s_{conv}, dans lequel toutes les requêtes effectuées après x_{quiet} sont effectuées. Comme seul un nombre fini de requêtes a été effectué avant x_{quiet}, H est *update* cohérent . □

Le compromis k Il y a deux manières de comprendre l'algorithme $UC[k]$. D'une part, il peut être vu comme un mécanisme de *garbage collection* ajouté

à l'algorithme UC_∞, qui permet l'élimination en toute sécurité des anciennes opérations. En ce sens, plus le k est petit, moins il faut de mémoire. D'autre part, l'algorithme $UC[k]$ peut être vu comme une optimisation de l'algorithme UC_0 dans lequel la liste des mises à jour $\mathsf{history}_i$ est utilisée comme cache pour stocker les nouvelles mises à jour tant qu'il existe un risque que de nouvelles mises à jour puissent être reçues, réduisant ainsi le besoin de diffuser des messages de correction. En ce sens, plus k est grand, moins de « défauts de cache » devraient se produire, et moins l'algorithme est gourmand en communication. Ainsi, la valeur k est un compromis entre l'espace mémoire utilisé et le nombre de messages envoyés.

Pour chaque exécution, il existe cependant une valeur minimale de k pour lequel aucun message `mCorrect` n'est envoyé. Toute valeur de k plus grande augmente les besoins en mémoire tout en conservant la complexité en communication constante, et toute valeur inférieure de k génère des conflits qui provoquent la diffusion de coûteux Messages `mCorrect`. Comme les horodatages de la liste des mises à jour $\mathsf{history}_i$, qui couvrent un intervalle de taille d'au plus k, sont générés à l'aide d'une horloge de Lamport qui encode la relation *happened-before*, cette valeur optimale correspond à la longueur de la plus longue chaîne causale d'opérations de mise à jour effectué pendant le transit d'un seul message `mUpdate`. Cette valeur dépend du délai de transmission des messages, à la fois par rapport aux autres messages et par rapport au temps entre deux mises à jour effectuées par un même processus.

La valeur de k a un impact important sur les performances de l'algorithme, mais le choix de la valeur parfaite nécessite des connaissances sur l'exécution future. Fait intéressant, k n'est pas utilisé dans la preuve d'exactitude. Une conséquence importante est que l'algorithme reste correct même si les processus ne partagent pas une valeur commune de k. En particulier, chaque processus peut mettre à jour sa valeur de k indépendamment, en fonction de ses propres caractéristiques telles que sa mémoire disponible, ou suivant une stratégie globale pour adapter k aux spécificités du système. Bien que la mise à jour d'un paramètre global soit très coûteuse, mettre à jour une valeur locale pour k est bon marché : augmenter k est gratuit, et diminuer k consiste uniquement à exécuter $\texttt{record}(\mathsf{time}_i - k)$ avec la nouvelle valeur de k.

6.4 Conclusion

Ce chapitre explore la complexité des constructions universelles *update* cohérentes dans des systèmes par passage de messages composés de processus asynchrones qui peuvent échouer en se bloquant. Pour pouvoir ordonner des opérations malgré l'asynchronie, UC_∞, une construction déjà proposée, stocke toutes les opérations dans un journal de plus en plus grand. Bien que cette construction soit optimale dans le nombre de messages échangés, sa complexité spatiale peut être prohibitive pour de nombreuses applications. Le but de ce chapitre est de réduire la complexité spatiale des constructions universelles *update* cohérentes, mais en gardant la complexité en nombre de messages aussi faible que possible.

La première contribution du chapitre est une preuve que le schéma de messages optimal adopté dans L'algorithme UC_∞, appelé modèle opérationnel, oblige à stocker $O(m)$ bits d'information pour une exécution contenant $O(m)$ opérations, ce qui est la même complexité asymptotique que l'algorithme UC_∞. En revanche, nous avons prouvé que des algorithmes de complexité spatiale logarithmique étaient disponibles dans le modèle plus général *wait-free*, au prix d'une plus grande complexité en terme de communication.

C'est un résultat théorique important, car il répond négativement à la question suivante : le modèle *wait-free* et le modèle opérationnel sont-ils équivalents en termes de complexité spatiale ? Il montre que la question de savoir si le modèle opérationnel est bien adapté pour représenter la tolérance de partition n'est pas simple, en particulier dans le contexte de l'étude de la complexité de la mémoire locale requise pour implémenter des objets partagés. Une question ouverte intéressante est de savoir si les bornes inférieures prouvées pour plusieurs objets dans le modèle opérationnel peuvent être étendues au modèle *wait-free*.

La deuxième contribution du chapitre est une nouvelle construction universelle *update* cohérente. Les performances de l'algorithme proposé dépendent d'un paramètre k qui ne peut être choisi parfaitement sans connaissance de l'exécution future. Une valeur trop faible de k peut rendre l'exécution coûteuse en termes de communication, alors qu'une valeur élevée de k nécessite plus de mémoire et de ressources de calcul. Un travail futur possible serait d'explorer les meilleures stratégies pour adapter dynamiquement k en fonction de certaines conditions de communication, afin d'optimiser les besoins en ressources.

Chapitre 7

Conclusion

7.1 Résumé

Dans les systèmes modernes tels que le *multi-threading* ou encore les modèles de Foglets utilisés dans le projet O'Browser, l'une des principales caractéristiques est la potentielle création de *threads* tout au long d'un exécution (ou l'arrivée de nouveaux participants dans les foglets), nécessitant de modifier les modèles classiques où le nombre de participants est fixé au début d'une exécution. De nouveaux modèles, appelés modèles ouverts ont ainsi été formalisés. Afin d'étendre les résultats de calculabilité à ces systèmes, nous étudions la faisabilité des constructions universelles *wait-free* et linéarisables à partir de différents objets, et étendons ainsi la fameuse hiérarchie de Herlihy aux systèmes ouverts. Pour répondre aux besoins des systèmes Foglets de O'Browser, nous proposons une construction universelle flexible permettant de proposer un compromis entre complexité en espace de mémoire locale et complexité en nombre de messages envoyés afin d'implémenter n'importe quel objet partagé en cohérence faible.

Les résultats des deux parties de cette thèse sont les suivants :

1. L'extension de la hiérarchie de Herlihy aux différents systèmes ouverts dans le cas du fonctionnement par mémoire partagée. Nous prouvons l'existence de chaque catégorie et montrons la séparations entre les différents degrés. Cette extension est montrée sur la figure 7.1.

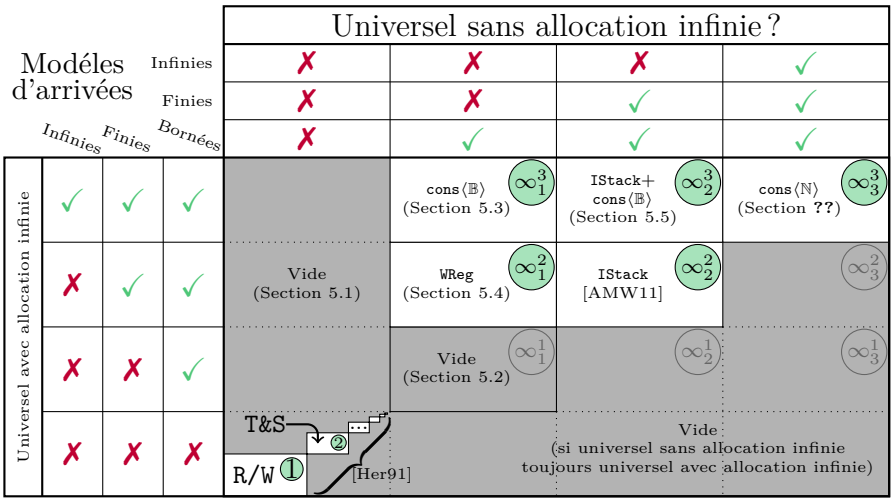

FIGURE 7.1 – Hiérarchie des modèles *wait-free* étendue

2. L'étude de la complexité des constructions universelles *wait-free* faiblement cohérentes dans le cas du fonctionnement par passage de message. Nous montrons que certains sous modèles permettent d'optimiser la complexité en nombre de messages envoyés, au détriment de la mémoire locale, et proposons une construction universelle qui, via un paramètre k, permet d'optimiser la complexité spatiale au prix d'une plus forte complexité de communication, et vis-versa, permettant ainsi de proposer un compromis.

7.2 Perspectives futures

Dans le chapitre 4 nous proposons une construction universelle dans M_3 à partir de registres et d'objets consensus. Le fonctionnement de notre construction nécessite que nous gardions en mémoire toutes les opérations appelées sur l'objet. Une question intéressante est de savoir s'il est possible de réduire le coût en mémoire en supprimant les opérations au bout d'un moment comme dans le cas de la construction $UC[k]$.

Dans le chapitre 5, nous étudions des objets, en particulier les registres fenêtrés et les piles d'itérateurs, qui sont complexes dans le sens où ils nécessitent généralement plusieurs emplacements de mémoire pour être mis en œuvre. Un autre problème ouvert est l'existence d'instructions spécifiques fonctionnant sur

un seul emplacement de mémoire, qui rempliraient chaque degré de la nouvelle hiérarchie.

De plus, l'objet de numéro de consensus ∞_2^3 que nous avons présenté est une composition d'un objet avec le numéro de consensus ∞_2^2 et d'un objet avec le numéro de consensus ∞_1^3. Cela amène deux questions :

1. Existe-t-il un objet de numéro de consensus ∞_2^3 qui ne peut pas être exprimé comme une telle composition ?
2. Inversement, existe-t-il deux objets de consensus numéro ∞_2^2 et ∞_1^3 dont la composition a le numéro de consensus ∞_3^3 ?

Dans le chapitre 6, nous prouvons que le modèle opérationnel n'est pas optimal dans le cadre de la complexité locale. Comme de nombreux résultats de complexité locale pour des objets en cohérence faible ont été montrés dans ce sous modèle, une question intéressante est de savoir s'ils pourraient être étendus au modèle *wait-free*.

Quant à la construction universelle $UC[k]$, il pourrait être intéressant d'explorer les meilleures stratégies pour adapter dynamiquement k, afin d'optimiser les besoins en ressources.

BIBLIOGRAPHIE

[AMW11] Yehuda AFEK, Adam MORRISON et Guy WERTHEIM, « From bounded to unbounded concurrency objects and back », in : *Proc. of the 30th Annual ACM Symposium on Principles of Distributed Computing, San Jose*, 2011, p. 119-128.

[Agu04] Marcos K AGUILERA, « A pleasant stroll through the land of infinitely many creatures », in : *ACM Sigact News* 35.*2* (2004), p. 36-59.

[ASS02] James ASPNES, Gauri SHAH et Jatin SHAH, « Wait-free consensus with infinite arrivals », in : *Proc. of the 34th ACM symposium on Theory of computing*, 2002, p. 524-533.

[ABD95] Hagit ATTIYA, Amotz BAR-NOY et Danny DOLEV, « Sharing memory robustly in message-passing systems », in : *Journal of the ACM (JACM)* 42.*1* (1995), p. 124-142.

[AEM17] Hagit ATTIYA, Faith ELLEN et Adam MORRISON, « Limitations of Highly-Available Eventually-Consistent Data Stores », in : *IEEE Trans. Parallel Distrib. Syst.* 28.*1* (2017), p. 141-155, DOI : 10.1109/TPDS.2016.2556669, URL : https://doi.org/10.1109/TPDS.2016.2556669.

[AW94] Hagit ATTIYA et Jennifer L WELCH, « Sequential consistency versus linearizability », in : *ACM Transactions on Computer Systems (TOCS)* 12.*2* (1994), p. 91-122.

[Att+90] Hagit ATTIYA et al., « Renaming in an asynchronous environment », in : *J. ACM* 37.*3* (1990), p. 524-548.

[Att+16] Hagit ATTIYA et al., « Specification and complexity of collaborative text editing », in : *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, ACM, 2016, p. 259-268.

[Bal+95] Roberto Baldoni et al., « Characterization of consistent global checkpoints in large-scale distributed systems », in : *Distributed Computing Systems, 1995., Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of (FTDCS'15)*, IEEE, 1995, p. 314-323.

[BMP19a] Grégoire Bonin, Achour Mostefaoui et Matthieu Perrin, « Does the operational model capture partition tolerance in distributed systems ? », in : *International Conference on Parallel Computing Technologies*, Springer, 2019, p. 400-407.

[BMP19b] Grégoire Bonin, Achour Mostéfaoui et Matthieu Perrin, « Wait-Free Universality of Consensus in the Infinite Arrival Model », in : *International Symposium on Distributed Computing (DISC)*, 2019.

[BT85] Gabriel Bracha et Sam Toueg, « Asynchronous consensus and broadcast protocols », in : *Journal of the ACM (JACM)* 32.*4* (1985), p. 824-840.

[Bur+14] Sebastian Burckhardt et al., « Replicated data types : specification, verification, optimality », in : *ACM SIGPLAN Notices*, t. 49, ACM, 2014, p. 271-284.

[CRR17] Armando Castañeda, Michel Raynal et Sergio Rajsbaum, « Long-Lived Tasks », in : *NETYS 2017 - 5th International Conference on NETworked sYStems*, t. 10299, Marrakech : Springer, 2017, p. 439-454.

[CPT15] Keren Censor-Hillel, Erez Petrank et Shahar Timnat, « Help ! », in : *Proc. of the ACM Symposium on Principles of Distributed Computing*, 2015, p. 241-250.

[CM05] Gregory Chockler et Dahlia Malkhi, « Active disk paxos with infinitely many processes », in : *Distributed Computing* 18.*1* (2005), p. 73-84.

[DeC+07] Giuseppe DeCandia et al., « Dynamo : amazon's highly available key-value store », in : *ACM SIGOPS Operating Systems Review*, t. 41, ACM, 2007, p. 205-220.

[DW15] Oksana Denysyuk et Philipp Woelfel, « Wait-freedom is harder than lock-freedom under strong linearizability », in : *International Symposium on Distributed Computing*, Springer, 2015, p. 60-74.

[Ell+16] Faith Ellen et al., « A complexity-based hierarchy for multiprocessor synchronization », in : *Proc. of the ACM Symposium on Principles of Distributed Computing*, 2016, p. 289-298.

[FK14] Panagiota Fatourou et Nikolaos D. Kallimanis, « Highly-Efficient Wait-Free Synchronization », in : *Theory Comput. Syst.* 55.*3* (2014), p. 475-520.

[FLP85] Michael J. Fischer, Nancy A. Lynch et Mike Paterson, « Impossibility of Distributed Consensus with One Faulty Process », in : *J. ACM* 32.*2* (1985), p. 374-382.

[GMT01] Eli Gafni, Michael Merritt et Gadi Taubenfeld, « The concurrency hierarchy, and algorithms for unbounded concurrency », in : *Proc. of the 20th ACM symposium on Principles of distributed computing*, 2001, p. 161-169.

[GL02] Seth Gilbert et Nancy Lynch, « Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services », in : *Acm Sigact News* (2002).

[Her88] Maurice Herlihy, « Impossibility and Universality Results for Wait-Free Synchronization », in : *Proc. of the 7th Annual ACM Symposium on Principles of Distributed Computing, Toronto*, 1988, p. 276-290.

[Her91] Maurice Herlihy, « Wait-free synchronization », in : *ACM TOPLAS* 13.*1* (1991), p. 124-149.

[HLM03] Maurice Herlihy, Victor Luchangco et Mark Moir, « Obstruction-free synchronization : Double-ended queues as an example », in : *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.* IEEE, 2003, p. 522-529.

[HRR13] Maurice Herlihy, Sergio Rajsbaum et Michel Raynal, « Power and limits of distributed computing shared memory models », in : *Theor. Comput. Sci.* 509 (2013).

[HS08] Maurice Herlihy et Nir Shavit, *The art of multiprocessor programming*, Morgan Kaufmann, 2008, isbn : 978-0-12-370591-4.

[HW90] Maurice Herlihy et Jeannette M. Wing, « Linearizability : A Correctness Condition for Concurrent Objects », in : *ACM TOPLAS* 12.*3* (1990), p. 463-492.

[Lam78] Leslie Lamport, « Time, clocks, and the ordering of events in a distributed system », in : *Communications of the ACM* 21.*7* (1978), p. 558-565.

[Lam79] Leslie Lamport, « How to make a multiprocessor computer that correctly executes multiprocess programm », in : *IEEE transactions on computers 9* (1979), p. 690-691.

[Lam+01] Leslie Lamport et al., « Paxos made simple », in : *ACM Sigact News* 32.*4* (2001), p. 18-25.

[LSP82] Leslie Lamport, Robert Shostak et Marshall Pease, « The Byzantine Generals Problem », in : *ACM Transactions on Programming Languages and Systems* 4.*3* (1982), p. 382-401.

[LS88] Richard J Lipton et Jonathan S Sandberg, *PRAM : A scalable shared memory*, Princeton University, Department of Computer Science, 1988.

[MT03] Michael Merritt et Gadi Taubenfeld, « Resilient consensus for infinitely many processes », in : *Proc. of the International Symposium on Distributed Computing*, 2003, p. 1-15.

[Mil+02] Dejan S Milojicic et al., *Peer-to-peer computing*, 2002.

[MPR18] Achour Mostefaoui, Matthieu Perrin et Michel Raynal, « A simple object that spans the whole consensus hierarchy », in : *Parallel Processing Letters* 28.*02* (2018).

[MR01] Achour Mostefaoui et Michel Raynal, « Leader-based consensus », in : *Parallel Processing Letters* 11.*01* (2001), p. 95-107.

[MRT00] Achour Mostefaoui, Michel Raynal et Frédéric Tronel, « From binary consensus to multivalued consensus in asynchronous message-passing systems », in : *Information Processing Letters* 73.*5-6* (2000), p. 207-212.

[PSL80] Marshall Pease, Robert Shostak et Leslie Lamport, « Reaching agreement in the presence of faults », in : *Journal of the ACM (JACM)* 27.*2* (1980), p. 228-234.

[PMB20] Matthieu Perrin, Achour Mostefaoui et Grégoire Bonin, « Extending the Wait-free Hierarchy to Multi-Threaded Systems », in : *Proceedings of the 39th Symposium on Principles of Distributed Computing*, 2020, p. 21-30.

[PMB21] Matthieu Perrin, Achour Mostéfaoui et Grégoire Bonin, « Extension de la Hiérarchie de Herlihy aux Systèmes Multi-threads », in : *ALGOTEL 2021—23èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications*, 2021.

[PMJ15] Matthieu Perrin, Achour Mostefaoui et Claude Jard, « Update consistency for wait-free concurrent objects », in : *Parallel and Distributed Processing Symposium (IPDPS'15), 2015 IEEE International*, IEEE, 2015, p. 219-228.

[PMJ16] Matthieu Perrin, Achour Mostefaoui et Claude Jard, « Causal consistency : beyond memory », in : *Proc. of the 21st ACM Symposium on Principles and Practice of Parallel Programming*, 2016, p. 1-12.

[Plo89] Serge A Plotkin, « Sticky bits and universality of consensus », in : *Proc. of the 8th ACM Symposium on Principles of distributed computing*, 1989, p. 159-175.

[RLT78] Brian Randell, Pete Lee et Philip C. Treleaven, « Reliability issues in computing system design », in : *ACM Computing Surveys (CSUR)* 10.*2* (1978), p. 123-165.

[Ray12] Michel Raynal, *Concurrent Programming - Algorithms, Principles, and Foundations*, Springer Science & Business Media, 2012.

[Ray13] Michel Raynal, *Distributed algorithms for message-passing systems*, t. 500, Springer, 2013.

[Ray16] Michel Raynal, « A look at basics of distributed computing », in : *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2016, p. 1-11.

[Ray17] Michel Raynal, « Distributed Universal Constructions : a Guided Tour », in : *Bulletin of the EATCS* 121 (2017).

[RST91] Michel Raynal, André Schiper et Sam Toueg, « The causal ordering abstraction and a simple way to implement it », in : *Information processing letters* 39.*6* (1991), p. 343-350.

[RD87] Andrew William Roscoe et Naiem Dathi, « The pursuit of deadlock freedom », in : *Information and Computation* 75.*3* (1987), p. 289-327.

[Sha+11] Marc Shapiro et al., « Conflict-free replicated data types », in : *Symposium on Self-Stabilizing Systems*, Springer, 2011, p. 386-400.

[Vog09] Werner Vogels, « Eventually consistent », in : *Communications of the ACM* 52.*1* (2009), p. 40-44.

[ZC09] Jialin Zhang et Wei Chen, « Bounded cost algorithms for multivalued consensus using binary consensus instances », in : *Information Processing Letters* 109.*17* (2009), p. 1005-1009.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE NANTES

Titre : Structures de Données pour environnements distribués à grande échelle

Mot clés : Systèmes distribués, Systèmes ouverts, Constructions universelles, Mémoire partagée, Systèmes par passage de message

Résumé : Dans les systèmes tels que les Foglets utilisés dans le projet O'Browser, ou les systèmes multi-thread modernes, de nouveaux processus peuvent arriver au cours de l'exécution : nous appelons cela les systèmes ouverts. Cela nous amène à étudier la puissance de synchronisation des objets distribués, et plus particulièrement la faisabilité des constructions universelles, sous cette nouvelle hypothèse. Cette thèse présente les résultats obtenus sur l'universalité du consensus et l'extension de la hiérarchie *wait-free* de Herlihy dans les systèmes ouverts, ainsi qu'une étude de complexité et une solution sur les constructions universelles faiblement cohérentes.

Title: Data Structures for Large Scale Distributed Environments

Keywords: Distributed Systems, Arrival Models, Universal Constructions, Shared Memory, Message Passing Systems

Abstract: In systems such as Foglets used in the O'Browser project, or modern multi-threaded systems, new processes can join during an execution: we call these open systems. This leads us to study the power of synchronization of distributed objects, and more particularly the feasibility of universal constructions, under this new hypothesis. This thesis presents the results obtained on the universality of consensus and the extension of the wait-free hierarchy in open systems, as well as a study of complexity and a solution on weakly consistent universal constructions.