Calculabilité et conditions de progression des objets partagés en présence de défaillances

Damien Imbs

N° d'ordre : 4511 ANNÉE 2012

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale Matisse

présentée par

Damien Imbs

préparée à l'unité de recherche IRISA – UMR 6074
Institut de Recherche en Informatique et Systèmes Aléatoires
ISTIC

Calculabilité et conditions de progression des objets partagés en présence de défaillances

Thèse soutenue à Rennes le 12 Avril 2012

devant le jury composé de :

Carole DELPORTE-GALLET
Prof. Paris 7 / présidente

Dominique MERY
Prof. Nancy / rapporteur

Franck PETIT
Prof. Paris 6 / rapporteur

Rachid GUERRAOUI
Prof. EPFL / examinateur

Panagiota FATOUROU
Prof. Ioannina / examinatrice

Petr KUZNETSOV
Senior Researcher TU Berlin / examinateur

Michel RAYNAL
Prof. Rennes 1 / directeur de thèse

Table des matières

Chapitre 1

Introduction

Processus et synchronisation Aux commencements de l'informatique, un processeur exécutait un seul programme à la fois et jusqu'à sa fin. Les équipements informatiques étaient très couteux à l'époque et devaient être partagés entre de nombreux utilisateurs ; ceux-ci devaient donc attendre la fin du programme en cours avant de pouvoir en commencer un autre. Ce mode de fonctionnement étant très contraignant, les créateurs de systèmes ont eu l'idée d'exécuter plusieurs programmes en même temps sur un seul processeur : il y avait un *processus* par programme, un seul processeur et le système se chargeait de gérer quel processus exécuter à chaque instant. Le système devait aussi gérer d'éventuels conflits d'accès aux données. L'idée de synchronisation dans les systèmes informatiques a donc fait son apparition avec les systèmes d'exploitation multi-processus mais mono-processeur.

Apparition des réseaux Une des grandes révolutions de l'informatique a été la connection d'ordinateurs différents, c'est-à-dire la création des réseaux informatiques. Des ordinateurs éloignés géographiquement ne pouvant partager la même mémoire physique, ils communiquent en échangeant des messages. Ils peuvent également échanger des messages avec d'autres entités physiques, par exemple des imprimantes. La connection de différents ordinateurs amène la possibilité de *collaboration* entre différents processeurs pour réaliser un objectif commun ; c'est ce qu'on appelle un *système distribué*. Quand ces processeurs communiquent par échange de message (comme c'est le cas à travers un réseau), on appelle celà un système distribué *à passage de message*.

Mémoire partagée Les besoins en puissance de calcul de certaines applications dépassant largement la capacité d'un seul processeur, l'idée de rassembler plusieurs processeurs dans une même machine a fait son chemin. Une première possibilité est de relier différentes machines mono-processeurs ayant chacune sa propre mémoire à l'aide d'un réseau. Pour obtenir de bonnes performances, une autre approche a aussi été utilisée : mettre en commun une seule mémoire physique, celle-ci pouvant être accédée par tous les processeurs de la machine. Quand les différents processeurs d'une telle machine sont utilisés pour réaliser un objectif commun, on appelle celà un système distribué *à mémoire partagée*. Les algorithmes pour les systèmes à mémoire partagée sont généralement plus simples et plus faciles à créer que ceux pour les systèmes à passage de message. On a donc créé des algorithmes permettant de simuler une mémoire partagée dans un système à passage de message.

Arrivée des processeurs multi-cœurs et généralisation des systèmes distribués Jusqu'à une période récente, les ordinateurs multi-processeurs étaient rares et étaient généralement réservés aux professionnels ayant de gros besoins en puissance de calcul. La puissance des processeurs augmentait régulièrement et la majorité des utilisateurs se satisfaisaient d'une machine mono-processeur. Il y a quelques années, les fabriquants de composants électroniques se sont heurtés à une barrière physique les empêchant d'augmenter la fréquence des processeurs ; pour continuer leur compétition commerciale acharnée, ils

ont donc dû trouver d'autres arguments à mettre en avant. La voie qu'ils ont prise a été la suivante : ils ont intégré plusieurs unités de calcul sur une même puce. On appelle une telle puce un *processeur multi-cœur*. Au fil des années, les multi-cœurs sont devenus la norme : on ne trouve plus les processeurs mono-cœur que dans les ordinateurs les moins puissants vendus actuellement. Toute machine multi-cœur est potentiellement un système distribué ; de plus, Internet est de plus en plus présent et est utilisé pour des systèmes de pair-à-pair (qui sont des systèmes distribués). On peut donc dire que les systèmes distribués se sont généralisés et qu'ils sont un aspect important de l'informatique de l'avenir.

Informatique parallèle et informatique distribuée L'étude des systèmes constitués de plusieurs unités de calcul ou *nœuds* se divise en deux branches principales : l'informatique parallèle et l'informatique distribuée. L'étude de l'informatique parallèle considère généralement que tous les nœuds ont la même entrée et se concentre donc sur la manière d'optimiser la répartition d'un travail nécessitant beaucoup de puissance de calcul. L'étude de l'informatique distribuée considère par contre que chaque nœud a une entrée propre, qui n'est pas connue des autres nœuds. L'informatique distribuée se concentre donc sur les problèmes de synchronisation et d'échange d'information entre les différents nœuds. Cela se concrétise par la recherche de ce qui peut et ne peut pas être atteint en termes de synchronisation dans un modèle de système donné, ainsi que la manière la plus efficace d'y arriver quand c'est possible. Dans la suite de ce document, comme il est d'usage dans la communauté de l'informatique distribuée, nous appellerons une unité de calcul un *processus* (à ne pas confondre avec les processus des systèmes d'exploitation).

Incertitude Par l'étude de la synchronisation, l'informatique distribuée cherche à maitriser l'incertitude inhérente à la mise en relation de différents processus déterministes. Le paradoxe des systèmes distribués est en effet qu'un ensemble d'éléments qui, considérés individuellement, sont totalement déterministes peut devenir non-déterministe.

Ce non-déterminisme a plusieurs causes :

- *L'asynchronie.* Le fait que les processus ne progressent pas à la même vitesse les empêche de savoir quelle partie de leur code les autres processus sont en train d'exécuter à un instant donné.
- *La présence de pannes.* Une panne peut empêcher la propagation d'information dans le système, voire même, pour certains types de panne, causer la propagation d'informations erronées. Associée à l'asynchronie, la possibilité de pannes empêche la distinction entre un processus défectueux et un processus qui est juste extrêmement lent.
- *L'ignorance des entrées et de l'état des autres processus du système.* L'entrée d'un processus n'est connue, au départ, que de lui seul. Il peut la communiquer aux autres éléments du système, mais cette communication peut être retardée à cause de l'asynchronie, voire ne pas avoir lieu si le processus tombe en panne. De même, un processus ne peut pas connaitre l'état d'un autre processus, sauf si ce dernier le lui communique explicitement.

Objets partagés Les problèmes de synchronisation (c'est-à-dire de maitrise de l'incertitude) dans les systèmes distribués sont généralement représentés par la spécification d'un objet partagé. Dans le cas où le problème peut être résolu dans un modèle donné, la solution consiste alors en une implémentation correcte (c'est-à-dire respectant les spécifications) de l'objet correspondant. Dans le cas contraire, on cherche à prouver qu'une telle implémentation n'existe pas (ce qui est généralement nettement plus difficile à obtenir).

Critères de correction Les critères de correction qui spécifient le comportement d'un objet partagé peuvent être classés en deux catégories : les *conditions de progression* et les *conditions de sûreté*. Les conditions de progression déterminent quand l'invocation d'une opération doit terminer, c'est-à-dire rendre un résultat et permettre au processus de continuer son exécution. Elles peuvent dépendre de la contention (nombre d'autres processus s'exécutant durant l'opération) ou du nombre de processus

défaillants, mais elles sont définies indépendamment de la nature de l'objet. Les conditions de sûreté définissent, pour chaque entrée possible de l'opération, un ensemble de sorties correctes. Contrairement aux conditions de progression, elles sont donc intimement liées à la nature de l'objet.

Types d'objets partagés Les objets partagés représentant les problèmes de synchronisations peuvent être classés en deux grandes catégories : les objets persistants (*long-lived*) et les objets de décision (*one-shot*). Les objets persistants offrent aux processus une ou plusieurs méthodes que ces derniers peuvent appeler autant de fois que nécessaire. Par contre, les objets de décision n'offrent qu'une seule méthode que chaque processus ne peut appeler qu'une seule fois.

Objets persistants Un objet persistant offre aux processus une abstraction qui leur permet de communiquer plus simplement qu'en utilisant les primitives de base du système. Un exemple classique d'objet persistant est le snapshot. Dans sa version classique (il existe des variantes), le snapshot se base sur un système dans lequel les processus peuvent communiquer à travers des registres mono-écrivain/multi-lecteurs, c'est-à-dire que chaque processus a un registre attitré, de taille non bornée, dans lequel il peut écrire et il peut lire le contenu de tous les autres registres du système. Ces opérations ont lieu de manière atomique, c'est-à-dire qu'elles apparaissent aux processus comme ayant lieu à un instant précis, et pas sur un intervalle de temps. Il y a donc un ordre total sur ces opérations ; elles ne peuvent pas se chevaucher dans le temps. Par contre, la lecture de plusieurs registres ne peut pas se faire de manière atomique ; elle doit se faire registre par registre. Un objet snapshot permet de rendre atomique la lecture de l'ensemble des registres du système.

Objets de décision et tâches Un objet de décision offre aux processus une opération unique, qui ne peut être appelée qu'une seule fois par chaque processus. L'objet de décision le plus connu (et peut-être le plus simple) est le *consensus*. Il offre une méthode qui prend un paramètre : la valeur proposée par le processus. Il renvoie une valeur (dite *valeur de décision*) qui doit satisfaire les conditions de sûreté suivantes :

- *Validité.* La valeur de décision a été proposée par un processus.
- *Accord.* Toutes les valeurs de décision sont identiques.

Dans la version classique du consensus, on requiert la condition de progression *sans-attente* : tous les processus corrects (c'est-à-dire qui ne sont pas victimes de pannes) doivent recevoir une valeur de décision (c'est la condition de progression la plus contraignante).

Un des résultats les plus importants et les plus connus dans l'histoire de l'étude des systèmes distribués est que, dans un système asynchrone dans lequel au moins un processus peut subir une panne, le consensus sans-attente est impossible : c'est le fameux résultat FLP [51] (du nom de Fischer, Lynch et Paterson, ses auteurs).

Les conditions de sûreté d'un objet de décision peuvent être modélisées par une *tâche*. Une tâche est définie par un ensemble $\mathcal{I}$ de vecteurs d'entrée (pour chaque vecteur, un élément par processus qui correspond à sa valeur proposée), un ensemble $\mathcal{O}$ de vecteurs de sortie (pour chaque vecteur, un élément par processus qui correspond à sa valeur de décision) et une relation Δ qui définit, pour chaque vecteur d'entrée I, un ensemble $\Delta(I)$ de vecteurs de sorties qui peuvent lui être associés.

Tâches colorées et non colorées On distingue deux types de tâches : les tâches *non colorées (colorless tasks)* et les tâches *colorées (colored tasks)*.

Une tâche est non colorée si pour chaque vecteur d'entrée I, (1) toutes les permutations de ce vecteur ont le même ensemble de sortie (c'est-à-dire que si I' est une permutation de I, alors $\Delta(I') = \Delta(I)$) et (2) toute valeur de décision d'un processus peut être choisie comme valeur de décision par n'importe quel autre processus. Le consensus est un exemple de tâche non colorée : (1) l'identité du processus qui propose une valeur n'a pas d'importance et (2) tous les processus doivent décider la même valeur.

Si une tâche ne satisfait pas la définition ci-dessus, elle est colorée. Un exemple de tâche colorée est le renommage : chaque processus a comme entrée une identité unique (deux processus ne peuvent pas avoir la même entrée) et doit décider une nouvelle identité unique parmi un nouvel espace d'indentité réduit (si le nouvel espace est $[1..M]$, on appelle cette tâche le M-renommage). Chaque sortie devant être unique, une valeur ne peut pas être décidée deux fois ; le renommage est donc une tâche colorée.

Conditions de progression et calculabilité L'étude de la calculabilité d'un objet dans un système donné consiste à déterminer, pour une condition de progression donnée, s'il existe une implémentation correcte de l'objet dans le système correspondant. En guise d'exemple, considérons le k-accord ensembliste sans attente. Le k-accord ensembliste est une généralisation du consensus : la propriété d'accord est modifiée pour devenir : "au plus k valeurs différentes peuvent être décidées". Dans un système asynchrone en mémoire partagée où les processus n'ont accès qu'à des registres et où au plus t processus peuvent subir une panne, le k-accord ensembliste peut être réalisé si et seulement si $k > t$.

Si on relache la condition de progression pour ne requérir la terminaison que dans les cas où le processus s'exécute sans concurrence après un temps fini (les autres processus sont soit en panne, soit retardent l'exécution de leur prochaine instruction), le k-accord ensembliste devient possible quelles que soient les valeurs de k et de t. Comme nous allons le voir tout au long de ce document, la calculabilité d'un objet dans un système donné est intimement liée à la condition de progression requise.

Contributions

Conditions de progression asymétriques Les conditions de progression classiques imposent les mêmes conditions à tous les processus sans distinction. Le chapitre 3 introduit la notion de conditions de progression *asymétriques* : des conditions de progression qui peuvent imposer différentes conditions de terminaison à différents processus. Il introduit ensuite la condition de progression (n, x)-*vivacité* et montre qu'elle définit une nouvelle hiérarchie stricte d'objets. Finalement, il introduit la condition de progression x-*sans-attente* qui permet de résoudre le problème du consensus (tous les processus doivent s'accorder sur une seule valeur proposée) dans un système dans lequel seul un sous-ensemble de processus doivent toujours décider une valeur.

Ces résultats ont été publiés dans [87, 88, 80].

Abstractions Le chapitre 4 considère des abstractions : des objets qui sont utilisés pour faciliter la conception d'algorithmes mais qui ne changent pas la calculabilité des objets dans le modèle. Il introduit d'abord un modèle de système hybride dans lequel l'ensemble des processus est partitionné, chaque partition ayant accès à une mémoire partagée qui n'est pas accessible aux processus qui ne font pas partie de cette partition. Les processus de partitions différentes peuvent communiquer en envoyant et en recevant des messages. Il montre ensuite que, dans ce modèle, une mémoire partagée globale ne peut pas être implémentée de manière sans-attente sans aide additionnelle et précise l'aide nécessaire et suffisante pour y parvenir.

L'abstraction du *snapshot* permet d'écrire dans un registre et de lire de manière atomique les valeurs de tous les registres du système (prendre un snapshot). Parfois, seules les valeurs d'un sous-ensemble de ces registres sont nécessaires. Un algorithme de snapshot *partiel* permet à un processus de choisir de ne lire qu'un sous-ensemble des registres. Quand on considère le modèle dans lequel les processus ont accès à une mémoire partagée globale, l'abstraction du snapshot ne renforce pas le système : elle peut être implémentée de manière sans-attente sans aide additionnelle. Durant une écriture, les algorithmes de snapshot classiques calculent d'abord l'ensemble des valeurs des registres du système, puis écrivent. Un nouvel algorithme est présenté qui écrit, puis calcule les valeurs des autres registres. De plus, cet algorithme permet de prendre un snapshot partiel de manière efficace (une opération d'écriture ne doit

calculer que les valeurs des registres requises par les opérations de snapshot qui sont concurrentes et qui lisent le registre écrit).

Ces résultats ont été publiés dans [83, 78, 91].

Simulations Une simulation permet de comparer différents modèles de systèmes en déterminant si un modèle peut être implémenté dans un autre modèle. Un travail à la base de ce domaine est la simulation *Borowsky-Gafni*. La simulation BG montre que, quand on considère un modèle dans lequel les processus n'ont accès qu'à des registres partagés, un objet peut être implémenté de manière t-résiliente (les processus doivent décider une valeur si au plus t d'entre eux crashent) si et seulement si l'objet peut être implémenté de manière sans-attente dans un système de $t+1$ processus. La simulation originale ne considère que des objets pour lesquels un processus peut décider n'importe quelle valeur déjà décidée par un autre processus ; elle a ensuite été étendue au cas général. Le chapitre 5 commence par présenter une manière alternative de réaliser la version étendue de la simulation BG. Il montre ensuite, en utilisant des simulations, qu'un objet peut être implémenté dans un système où les processus ont accès au x-consensus (tous les sous-ensembles d'au plus x processus peuvent résoudre le consensus) et où au plus t processus peuvent crasher si et seulement si le même objet peut être implémenté dans un système où les processus n'ont accès qu'à des registres partagés et où au plus $\lfloor t/x \rfloor$ processus peuvent crasher.

Ces résultats ont été publiés dans [90, 85].

Cassage de la symétrie Le chapitre 6 considère les tâches de *cassage de la symétrie* : des tâches dans lesquelles des processus qui démarrent dans le même état, à part un identifiant unique mais non borné, doivent décider une valeur bornée. Les conditions sur cette valeur dépendent de la tâche, la condition minimale étant que tous les processus ne décident pas la même valeur. Une tâche de cassage de la symétrie bien connue est le M-renommage, dans laquelle tous les processus doivent décider une valeur différente prise dans $[1..M]$. Dans la version *adaptative* du renommage, le nombre p de processus participants n'est pas connu et M est une fonction de p. La condition de progression x-sans-obstruction requiert qu'un processus décide une valeur en un temps fini si, après un instant t, au plus x processus exécutent leur programme. Un algorithme qui résout le $(p+x-1)$-renommage adaptatif de manière x-sans-obstruction est présenté.

Finalement, le concept de tâches de *cassage de la symétrie généralisé* (tâches GSB, *Generalized Symmetry Breaking tasks*) est présenté. Ce concept unifie dans un seul cadre de nombreux problèmes qui étaient considérés auparavant comme indépendants, tels que le renommage non-adaptatif et le cassage de symétrie faible (*Weak Symmetry Breaking*). Une tâche GSB est définie par quatre paramètres : le nombre n de processus qui peuvent accéder à la tâche, le nombre m de valeurs de sortie possibles et les bornes minimale ℓ et maximale u sur le nombre de processus qui peuvent décider chaque valeur. Par exemple, la tâche GSB $\langle n, n, 1, 1\rangle$ correspond au renommage parfait dans lequel la taille de l'espace des noms correspond exactement au nombre de processus dans le système. Pour chaque n, la famille de tâches GSB $\langle n, -, -, -\rangle$ peuvent être ordonnées partiellement selon leurs vecteurs de sortie. Après la présentation de différents résultats caractérisant la calculabilité et la puissance relative des tâches GSB, il est montré que la tâche GSB $\langle n, n, 1, 1\rangle$ (renommage parfait) est strictement plus fort que le $(n-1)$-accord ensembliste dans lequel n processus doivent s'accorder sur au plus $n-1$ valeurs proposées).

Ces résultats ont été publiés dans [89, 82, 34].

Autres contributions Durant ce travail de thèse, d'autres contributions ont été apportées. Par souci de cohérence, elles ne sont pas présentées en détails dans ce manuscrit. Dans [79, 93], un nouveau critère de correction a été défini pour les *mémoires transactionnelles*. Ce critère est plus faible que ceux définis précédemment, tout en gardant l'esprit des garanties qu'ils offrent. Un algorithme qui respecte ce nouveau critère est également présenté. Dans [41, 42, 81, 92], d'autres nouveaux algorithmes de mémoires

transactionnelles ont été présentés. Dans [94], un lien est fait entre les primitives de synchronisation LL/SC et la notion de *registre temporisé*.

Collaborations Tous les travaux présentés dans ce manuscrit ont été réalisés en collaboration avec Michel Raynal. Les travaux présentés dans la section 3.1 ont été réalisés en collaboration avec Gadi Taubenfeld. Les travaux présentés dans la section 6.2 ont été réalisés en collaboration avec Sergio Rajsbaum et Armando Castañeda.

Chapitre 2

Contexte de l'étude

2.1 Modèles de systèmes

2.1.1 Processus

On considère un système constitué d'un ensemble fini de processus $\Pi = \{p_1, \ldots, p_n\}$. Chaque processus p_i possède un index $i : 1 \leq i \leq n$ mais cet index n'est pas nécessairement connu de p_i. Le processus p_i exécute une suite d'instructions définie par son programme. Chaque processus est *séquentiel* : il exécute ses instructions une après l'autre. Un processus ne peut donc pas exécuter plusieurs instructions en parallèle.

Le processus p_i possède une identité unique id_i. Si pour chaque p_i, $id_i = i$, chaque processus connait les identités de tous les processus du système. Dans certains cas (qui seront précisés), $id_i \neq i$; p_i ne connait alors que sa propre identité et ne peut apprendre celle des autres processus qu'en communiquant (directement ou indirectement) avec eux. Pour simplifier l'écriture des algorithmes, les indexes des processus pourront quand même être utilisés dans un but d'adressage (par exemple pour lire les données écrites dans un tableau), mais on interdira l'utilisation directe de ces indexes dans le code des algorithmes.

On considère ici des processus asynchrones : le temps entre l'exécution de deux instructions d'un processus correct est fini mais non borné et n'est pas connu à l'avance. On ne peut donc pas faire d'hypothèse sur les vitesses d'exécution des processus. Les instructions des différents processus sont alors ordonnées selon un temps logique (par opposition au temps physique) : une instruction peut avoir été exécutée avant ou après une autre instruction, mais la différence de temps physique entre les instants d'exécution de ces deux instructions n'est pas pertinente. Dans un algorithme conçu pour fonctionner dans un système asynchrone, on ne peut donc faire aucune référence explicite au temps physique.

2.1.2 Pannes

Dans un système constitué d'un grand nombre d'entités, il est raisonnable de supposer qu'un certain nombre d'entre elles peuvent subir des pannes. Nous considérons ici le cas du *crash* (ou *panne franche*) : un processus peut arrêter son exécution à un endroit quelconque de son programme. Il n'exécute alors plus aucune instruction. Un processus qui crashe est dit *fautif*, sinon il est dit *correct*. Une conséquence d'un crash est que le processus cesse toute communication avec les autres processus du système.

Dans un système synchrone ou partiellement synchrone, on peut détecter un crash : si le processus ne répond pas dans un temps borné, on peut déterminer avec certitude qu'il a crashé. Dans un système asynchrone, cas dans lequel nous nous plaçons pour cette étude, on ne peut placer aucune borne temporelle entre les exécutions de deux instructions. Si un processus est crashé (et n'exécute donc plus d'instructions de communication), les autres processus du système ne peuvent pas déterminer avec certitude s'il est crashé ou s'il est simplement lent et reprendra son exécution. Le crash est donc particulièrement délicat

à traiter dans un système asynchrone : on doit toujours prendre en compte le fait qu'un processus qui semble crashé (car il ne répond pas) peut reprendre son exécution.

Pour résoudre certains problèmes, on peut avoir besoin de poser des hypothèses sur le nombre de crash du système. On dira alors qu'il y a au plus $t, 0 \leq t < n$ crashes dans le système, mais sans aucune hypothèse sur les identités des processus qui peuvent crasher.

2.1.3 Sous-systèmes de communication

Pour constituer un système distribué, les processus doivent pouvoir communiquer. Les modèles de communication entre processus les plus utilisés sont le *passage de messages* et la *mémoire partagée*. Dans un système à passage de messages, les processus communiquent en échangeant des messages à travers un réseau. Dans un système en mémoire partagée, les processus communiquent en utilisant des objets partagés : registres, piles, *Test&Set*, etc.

2.1.3.1 Communication par passage de message

Dans ce modèle, les processus communiquent en échangeant des messages à travers des canaux de communication. Il représente le mode de communication de base utilisé dans un réseau. Les processus disposent de deux opérations atomiques : $\mathsf{send}()$ et $\mathsf{receive}()$. L'opération $\mathsf{send}(j, m)$ envoie le message m au processus p_j. L'opération $\mathsf{receive}()$ renvoie le premier message de la file d'attente du canal.

On considère que les canaux sont fiables : aucun message n'est corrompu, perdu ou dupliqué. On ne fait par contre aucune hypothèse sur l'ordre de réception des messages, ni sur le délai de transmission d'un message.

Pour simplifier l'écriture des algorithmes, on considère que les processus ont accès à une opération $\mathsf{broadcast}()$. L'opération $\mathsf{broadcast}(m)$ envoie à tous les processus le message m. Cette opération se basant sur $\mathsf{send}()$, elle n'est pas atomique ; un processus qui crashe durant l'exécution de $\mathsf{broadcast}()$ peut envoyer le message à tous les processus, à aucun processus ou à un sous-ensemble des processus du système.

2.1.3.2 Communication par mémoire partagée

C'est dans ce modèle que se place la plus grande partie de cette étude.

Les processus communiquent en utilisant une mémoire partagée. Cette mémoire est constituée de registres et dans certains cas d'objets plus puissants comme des piles, des objets *Test&Set*, etc... On considère ici que ces objets de base sont atomiques [77].

Registres atomiques Les registres peuvent être de deux types : 1WMR (pour *single-writer/multi-reader*, mono-écrivain/multi-lecteurs) ou MWMR (pour *multi-writer/multi-reader*, multi-écrivains/multi-lecteurs).

Un registre 1WMR est associé à un processus p_i qui peut lui affecter une valeur. Les autres processus du système peuvent lire cette valeur mais ne peuvent pas la modifier. Un registre MWMR n'est associé à aucun processus : tous les processus du système peuvent lui affecter une valeur et lire son contenu.

Objets plus complexes En plus des registres, les processus peuvent avoir accès à des objets partagés plus complexes. Ces objets peuvent être des piles, des files ou des objets *RMW* (*Read-Modify-Write*, Lire-Modifier-Écrire).

Les objets RMW ont un état (la valeur de l'objet) et offrent en général une seule opération. Cette opération modifie l'état de l'objet et renvoie son ancienne valeur de manière atomique. Un exemple bien

connu est le *Test&Set*. Sa spécification est la suivante (initialement, la valeur de l'objet Test&Set X est $true$) :

$$X.\mathsf{Test\&Set}() : atomic\{tmp \leftarrow X; X \leftarrow false.\}$$

Un objet Test&Set permet par exemple de choisir un processus parmi plusieurs qui sont en compétition sur une même ressource.

2.2 Problèmes et protocoles

Les problèmes que nous considérons ici peuvent être modélisés sous la forme d'objets partagés. Le problème prend alors la forme d'une spécification de l'objet correspondant. Une solution au problème est une implémentation correcte de cet objet dans un système donné.

Les objets partagés peuvent être utiles pour enrichir un système donné avec des abstractions permettant de faciliter l'écriture des algorithmes (par exemple, l'implémentation de registres dans un système à passage de message permet d'utiliser des algorithmes basés sur une mémoire partagée dans un tel système).

Ils peuvent être classés en deux grandes catégories : les objets *persistants*, offrant une ou plusieurs opérations qui peuvent être appelées plusieurs fois par les processus, et les objets *de décision*, offrant une seule opération ne pouvant être appelée qu'une seule fois par chaque processus.

La spécification d'un objet partagé peut être divisé en deux critères : la *sûreté* et la *vivacité*. Les critères de sûreté concernent les valeurs de retour des opérations de l'objet : ils définissent le type de l'objet en spécifiant si une exécution donnée est correcte. Les critères de vivacité, aussi appelés *conditions de progression*, définissent dans quelles conditions une opération doit terminer. Ils sont indépendants du type de l'objet considéré.

2.2.1 Conditions de progression

Terminaison sans-attente La condition de progression la plus forte est la terminaison sans-attente (*wait-free*) : sous cette condition, les opérations d'un objet invoquées par un processus correct doivent toujours terminer, indépendamment du nombre de crash dans le système et de la concurrence.

Dans un système en mémoire partagée dans lequel les processus n'ont accès qu'à des registres, cette condition est parfois trop forte ; on ne peut par exemple pas implémenter un objet Test&Set sans-attente en n'utilisant que des registres [69]. Il est donc parfois utile de considérer des conditions de progression plus faibles, même si dans certains cas extrêmes elles ne garantissent pas la terminaison.

t-résilience Une de ces conditions plus faibles est la *t-résilience*. Les opérations d'un objet t-résilient doivent toujours terminer si au plus t processus crashent, t étant un paramètre de la condition.

Un exemple de cas où cette condition est utile est la construction d'un registre dans un système à passage de message. On ne peut pas construire un registre dans un système asynchrone à passage de message si $t \geq n/2$ [16], mais cela devient possible si $t < n/2$ [16, 14].

Terminaison sans-obstruction [71] La t-résilience affaiblit le critère de progression (par rapport à la terminaison sans attente) en réduisant le nombre de crashes tolérés. Une autre approche consiste à forcer les processus à terminer leurs opérations uniquement en l'absence de concurrence. C'est l'approche de la terminaison *sans-obstruction*.

La terminaison sans-obstruction force un processus à terminer son opération uniquement quand il est le seul à s'exécuter (donc sans concurrence) durant un temps *suffisamment long*. "Suffisamment long" n'est pas une durée de temps définie ; en des termes plus précis, un processus qui se trouve seul à

s'exécuter (par exemple si tous les autres processus du système ont crashé) doit terminer son opération en un nombre fini d'instruction.

Cette propriété ne dit rien sur le début de l'exécution ; un processus peut avoir été en concurrence avec d'autres avant un instant t auquel les autres processus ont tous arrêté leur exécution. Elle force la terminaison si, après cet instant t, aucun autre processus ne reprend son exécution jusqu'à la terminaison de l'opération par le processus considéré. On peut remarquer que, contrairement à la t-résilience, les crashes sont souhaitables pour atteindre cette condition de progression.

Terminaison x-sans-obstruction [117] La terminaison x-sans-obstruction est une généralisation de la terminaison sans-obstruction. Elle force un processus à terminer son exécution si au plus x processus (dont le processus considéré) s'exécutent en concurrence durant un temps suffisamment long.

2.2.2 Objets persistants

Une manière courante de définir le critère de sûreté d'un objet persistant est de (1) donner une spécification *séquentielle* de l'objet, définissant le comportement de l'objet en l'absence de concurrence. Les opérations de l'objet doivent alors apparaître comme ayant été exécutées instantanément. Elles peuvent ensuite être ordonnées totalement. Il ne reste plus qu'à (2) définir quels sont les ordres totaux acceptables.

Sérialisabilité [107] Un objet est *sérialisable* si (1) chacune de ses opération apparait comme ayant été exécutée instantanément et (2) l'ordre total des opérations respecte l'ordre des invocations de chaque processus. Ce critère a été introduit initialement pour les opérations des bases de données [107].

Linéarisabilité [77] Un objet est *linéarisable* si chacune de ses opérations apparait comme ayant été exécutée à un instant situé entre son invocation (en pratique, le premier accès par le processus à un objet partagé durant son exécution de l'opération) et sa fin (l'instruction $\mathsf{return}()$). Ce critère sera utilisé pour tous les objets partagés considérés dans cette étude.

On peut remarquer que, comme chaque processus est séquentiel, cette propriété implique que l'ordre des invocations de chaque processus est respecté (propriété (2) de la sérialisabilité).

Le snapshot [1] Un exemple d'objet persistant est le *snapshot* en mémoire partagée (à distinguer du snapshot en passage de messages [40]). Il sera utilisé à de nombreuses reprises dans ce document. Nous considérons ici la version multi-écrivains/multi-lecteurs.

Un objet snapshot multi-écrivains/multi-lecteurs consiste en m entrées (chacune étant un registre atomique multi-écrivains/multi-lecteurs) qui fournit aux processus deux opérations $\mathsf{update}()$ et $\mathsf{snapshot}()$ telles que :

- $\mathsf{update}_i(r, v)$ est invoquée par p_i pour écrire la valeur v dans l'entrée r ($1 \leq r \leq m$) de l'objet snapshot. Cette opération renvoie la valeur de contrôle ok.
- $\mathsf{snapshot}()$ permet à un processus d'obtenir la valeur de chaque entrée de l'objet. Elle renvoie une séquence correspondante de valeurs $\langle v_1, \cdots, v_m \rangle$.

Un objet snapshot est défini par les propriétés suivantes.

- Terminaison sans-attente. Chaque invocation $\mathsf{update}()$ ou $\mathsf{snapshot}()$ faite par un processus correct termine.
- Linéarisabilité. Les opérations invoquées par les processus (sauf éventuellement la dernière opération invoquée par un processus fautif [1]) apparaissent comme si elles avaient été exécutées l'une après l'autre, chacune ayant été exécutée à un instant situé entre son premier évènement et son dernier évènement.

1. Si une telle opération n'apparait pas dans la séquence, elle apparait comme n'ayant pas été invoquée.

Dans le cas du snapshot, la linéarisabilité signifie qu'une opération snapshot() renvoie toujours des valeurs qui étaient simultanément présentes en mémoire et qui sont à jour.

Le snapshot peut être construit en mémoire partagée sans aide additionnelle [1, 23]. Il permet d'abstraire une partie du non-déterminisme du système en permettant la lecture atomique d'un ensemble de registres.

2.2.3 Objets de décision

Le critère de sûreté d'un objet de décision peut être spécifié par une *tâche* [103]. Cette tâche définit quelles sorties sont acceptables en fonction des entrées. Plus formellement, une tâche $(\mathcal{I}, \mathcal{O}, \Delta)$ consiste en un ensemble de *vecteurs d'entrée* $\mathcal{I}$, un ensemble de *vecteurs de sortie* $\mathcal{O}$ et une relation Δ qui associe à chaque $I \in \mathcal{I}$ au moins un $O \in \mathcal{O}$. Tous les vecteurs sont de dimension n.

L'entrée $I[i]$ d'un vecteur d'entrées I représente l'entrée du processus p_i. Si p_i crashe avant le début de son exécution (c'est-à-dire avant son premier accès à la mémoire partagée), $I[i] = \perp$. De même, l'entrée $O[i]$ d'un vecteur de sorties O représente la sortie du processus p_i. Si p_i crashe avant de décider une valeur, $O[i] = \perp$. L'ensemble des vecteurs de sorties acceptables pour un vecteur d'entrée I est noté $\Delta(I)$.

Un algorithme $\mathcal{A}$ est donc une implémentation correcte d'un objet spécifié par la tâche $(\mathcal{I}, \mathcal{O}, \Delta)$ si, dans chaque exécution de $\mathcal{A}$ dont les entrées forment le vecteur I et dont les sorties forment le vecteur O, on a $O \in \Delta(I)$.

Consensus La tâche la plus connue est le problème du consensus [51], mentionné dans l'introduction. Ce problème est spécifié de la manière suivante :

- *Validité.* La valeur de décision a été proposée par un processus.
- *Accord.* Toutes les valeurs de décision sont identiques.

Dans ce cas, toutes les entrées de chaque vecteur de sortie O doivent contenir soit la même valeur v, soit $\perp$ (si le processus correspondant a crashé). De plus, si $O \in \Delta(I)$, v doit être une valeur présente dans I.

Le consensus a une place particulière dans l'étude des objets distribués. En effet, c'est un objet *universel* : tout objet partagé qui a une spécification séquentielle (c'est-à-dire la quasi-totalité des objets partagés) peut être implémenté de manière sans-attente dans un système dans lequel le consensus peut être résolu [69].

Un des résultats fondamentaux de l'informatique distribuée est que, dans un système dans lequel les processus n'ont pas accès à des objets plus puissants que des registres, le consensus ne peut pas être résolu de manière t-résiliente pour $t \geq 1$, c'est-à-dire si au moins un processus du système peut crasher [51, 101, 69].

Accord ensembliste L'accord ensembliste est une généralisation du consensus [37]. Dans le k-accord ensembliste, la propriété de validité est la même que celle du consensus mais la propriété d'accord est relâchée de la manière suivante :

- *Accord.* Au plus k valeurs différentes sont décidées.

Le k-accord ensembliste autorise donc les processus à s'accorder sur k valeurs différentes au lieu d'une seule. Dans un système dans lequel les processus n'ont accès qu'à des registres, il peut être résolu de manière t-résiliente si $t < k$, mais est impossible si $t \geq k$ [27, 75, 112].

Tâches colorées Dans les tâches du consensus et de l'accord ensembliste, une valeur décidée par un processus peut ensuite être décidée par n'importe quelle autre processus. C'est ce qu'on appelle une tâche *non colorée (colorless)*. Dans ces tâches, si on a $O \in \Delta(I)$ pour des vecteurs O et I donnés, on a donc également $O' \in \Delta(I)$ pour chaque vecteur O' formé à partir d'un sous-ensemble des valeurs de O. Si une tâche $\mathcal{T}$ ne remplit pas cette condition, la tâche est dite *colorée (colored)*.

Renommage Un exemple de tâche colorée est le problème du M-renommage (M*-renaming)*. Dans ce problème, l'entrée d'un processus p_i est une identité id_i unique mais non bornée, et il doit décider une nouvelle identité également unique mais prise dans $[1..M]$. Pour rendre le problème non trivial, l'index i de p_i ne peut pas être utilisé autrement que pour des fins d'adressage et les programmes de tous les processus doivent être identiques.

Plus formellement, le problème du renommage est spécifié par les propriétés suivantes :
- *Accord*. Aucune valeur n'est décidée par plus d'un processus.
- *Validité*. La nouvelle identité appartient à $[1..M]$.
- *Indépendance de l'index*. Le comportement d'un processus est indépendant de son index.

Renommage adaptatif Un algorithme de renommage est *adaptatif* si la taille M de l'espace des nouvelles identités ne dépend que du nombre p de processus qui demandent une nouvelle identité (et pas du nombre total n de processus dans le système). Plusieurs algorithmes de renommage adaptatif sans-attente ont été proposés tels que la taille de l'espace des nouvelles identités est $M = 2p - 1$ (par exemple [28]).

Nombre d'accord ensembliste d'une tâche La notion de *nombre d'accord ensembliste* d'une tâche de décision T a été introduite dans [58]. Il s'agit du plus grand entier k tel que T peut être résolue de manière sans-attente en utilisant des registres et des objets k-accord ensembliste. Une tâche T ayant comme nombre d'accord ensembliste k ne peut donc pas être utilisée pour résoudre une autre tâche T' ayant comme nombre d'accord ensembliste $k - 1$ (quand les processus ont, hormis une solution de la tâche T, uniquement accès à des registres).

2.2.4 Nombre de consensus d'un objet

La notion de *nombre de consensus (consensus number)* d'un objet partagé a été introduite dans [69]. Le nombre de consensus x d'un objet O est défini comme le nombre maximal de processus qui, en n'ayant accès qu'à des registres partagés et des instances de O, peuvent résoudre le problème du consensus. Si ce nombre maximal n'existe pas (le consensus peut être résolu pour n'importe quel nombre de processus), l'objet O a le nombre de consensus $+\infty$. Un objet ayant le nombre de consensus x est donc universel dans un système de $n \leq x$ processus et ne l'est pas dans un système de $n' > x$ processus.

Cette notion permet d'établir une hiérarchie sur les objets partagés en fonction de leur nombre de consensus. Ainsi, les registres partagés ont comme nombre de consensus 1, les objets Test&Set ainsi que les files et les piles ont le nombre de consensus 2 et Compare&Swap, qui peut résoudre le consensus pour un nombre quelconque de processus, a le nombre de consensus $+\infty$ [69].

2.2.5 Simulation d'un système dans un autre : la simulation BG

Étant donnés deux modèles de systèmes différents, il peut être intéressant de les comparer, c'est-à-dire de déterminer si tous les problèmes qui peuvent être résolus dans le premier système peuvent l'être également dans le second. Dans le cas de systèmes dans lesquels les processus n'ont accès qu'à des snapshots [2], la simulation BG (du nom de ses auteurs, Borowsky et Gafni) a permis de montrer qu'une tâche non colorée peut être résolue de manière t-résiliente dans un système de n processus (avec $n > t$) si et seulement si elle peut être résolue de manière sans-attente dans un système de $t + 1$ processus [27, 30].

Pour atteindre ce but, chacun des n processus du système de base simule l'ensemble des n' processus du système simulé. Les simulateurs s'accordent ensuite sur les entrées des processus simulés et sur les résultats de leurs opérations en mémoire partagée en utilisant des snapshots (réalisables avec des registres

2. Rappelons qu'un objet snapshot peut être réalisé de manière sans-attente dans un système où les processus n'ont accès qu'à des registres partagés.

partagés). La simulation est conçue de manière à ce qu'un crash d'un simulateur ne puisse entrainer le blocage que d'un seul processus simulé. Le fonctionnement de cette simulation est détaillé dans le chapitre 5.

La simulation BG originale ne traite que le cas des tâches non colorées. La simulation a été étendue au cas des tâches colorées dans [55].

Chapitre 3

Conditions de progression asymétriques

Les conditions sans-attente et sans-obstruction sont toutes les deux des conditions de progression symétriques : un processus ou un ensemble de processus n'est pas favorisé par rapport aux autres. Tous les processus sont égaux du point de vue de la progression. La principale différence est que la condition sans-obstruction dépend de la concurrence, ce qui n'est pas le cas avec la condition sans-attente.

Conditions de progression asymétriques Ce chapitre introduit et étudie la notion de *condition de progression asymétrique*. Cette étude a deux motivations. La première vient de l'observation que, dans certaines applications, certains processus sont plus importants que d'autres du point de vue de la vivacité. Ils doivent donc avoir des garanties de progression plus fortes quand ils accèdent certains objets. De plus, les processus les plus importants pour un objet donné ne seront pas forcément les plus importants pour un autre objet. La deuxième motivation est d'ordre théorique : comprendre les possibilités offertes par les conditions de progression asymétriques et leurs limitations peut nous permettre de mieux comprendre la nature profonde des conditions de progression et ce qu'elles permettent et ne permettent pas.

3.1 (y, x)-vivacité

Nous nous plaçons dans un modèle de système en mémoire partagée classique avec n processus asynchrones. Un objet (y, x)-vivace est un objet qui (1) peut être accédé par $y \leq n$ processus et qui (2) garantit la condition sans-attente pour un sous-ensemble prédéfini de de $x \leq y$ processus et la condition sans-obstruction pour les $y - x$ processus restants. L'entier y définit la *taille* de l'objet, tadis que x définit son *degré de vivacité*. Si $x = y = n$, l'objet est un objet sans-attente classique.

On peut observer que, quand on considère l'éventail de conditions entre sans-obstruction et sans-attente, un objet de consensus $(n, 1)$-vivace est le premier objet plus fort qu'un objet de consensus sans-obstruction. Plus généralement, d'un point de vue théorique, la question fondamentale est la caractérisation de la puissance des objets (y, x)-vivaces. Cette section s'intéresse aux questions suivantes.

- On peut construire un objet de consensus $(n, 0)$-vivace (c'est-à-dire un objet sans-obstruction) uniquement à partir de registres atomiques [71]. Que se passe-t-il si on requiert la condition sans-attente pour un seul processus et qu'on a accès à des objets de consensus sans-attente accessibles par seulement $(n - 1)$ processus ? Exprimé autrement, peut-on construire des objets de consensus $(n, 1)$-vivaces à partir d'objets de consensus $(n - 1, n - 1)$-vivaces et de registres atomiques ?
- Quel est le nombre de consensus d'un objet de consensus (y, x)-vivace ?

La première question peut être reformulée de la façon suivante : le consensus sans-attente pour $n - 1$ processus est-il suffisamment puissant pour construire un objet de consensus pour n processus où la condition sans-attente n'est requise que pour un seul processus, et seulement la condition sans-obstruction pour les autres ? La deuxième question concerne la puissance intrinsèque des objets (y, x)-vivaces.

Cette section commence par répondre à la première question en montrant un résultat d'impossibilité : on ne peut pas construire un objet de consensus $(n, 1)$-vivace en n'utilisant que des registres et des objets de consensus sans-attente pour $n - 1$ processus. Elle répond ensuite à la deuxième question en montrant qu'un objet de consensus (n, x)-vivace a comme nombre de consensus $(x + 1)$ et établit donc une hierarchie pour la (n, x)-vivacité.

3.1.1 Modèle de système utilisé

Le système est un système classique de processus asynchrones communiquant par mémoire partagé comme défini dans le chapitre 2. Il est constitué de n processus asynchrones notés $p_1, \ldots, p_n$ (ils seront parfois appelés p ou q). Un processus exécute une séquence d'étapes définie par son algorithme. Il exécute son algorithme correctement jusqu'au moment où il crashe (s'il crashe). Après avoir crashé, un processus n'exécute plus aucune étape. Dans une exécution, un processus qui crashe est dit *fautif*. Sinon, il est dit *correct*.

Les processus communiquent en utilisant des registres atomiques classiques (que chaque processus peut lire ou écrire) et des objets de consensus (y, x)-vivaces. Une paire d'ensemble de processus (Y, X) telle que $|Y| = y$, $|X| = x$ et $X \subseteq Y$ est associée à chaque objet de consensus (y, x)-vivace. Seuls les processus de Y peuvent accéder à l'objet. L'objet offre une seule opération propose() (uniquement aux processus de Y). S'il accède à l'objet, un processus ne peut invoquer l'opération qu'une seule fois ; il lui passe en paramètre la valeur qu'il propose au consensus. Toute invocation de l'objet qui termine renvoie une valeur.

Les propriétés d'un objet de consensus (y, x)-vivace sont les suivantes :

- *Validité*. Une valeur décidée a été proposée par un processus.
- *Accord*. Tous les processus qui décident une valeur décident la même valeur.
- *Terminaison*.
  - Terminaison sans-attente. Toute invocation par un processus correct de X termine.
  - Terminaison sans-obstruction. Toute invocation par un processus correct de $Y \setminus X$ termine si le processus s'exécute sans concurrence pendant une période suffisamment longue.

On peut observer qu'un objet de consensus (n, n)-vivace est un objet de consensus sans-attente classique, tandis qu'un objet de consensus $(n, 0)$-vivace est un objet de consensus sans-obstruction.

Remarque Dans le problème du consensus, selon la propriété d'accord, dès qu'une valeur a été décidée par un processus, tous les autres processus peuvent décider cette valeur.

3.1.2 Deux résultats d'impossibilité

Cette section démontre deux résultats d'impossibilité. Le premier répond de façon négative à la première question posée : un objet sans-attente pour un processus et sans-obstruction pour tous les autres ne peut pas être construit même si, en plus de registres atomiques, on dispose des objets les plus puissants dans un système de $n - 1$ processus : les objets de consensus sans-attente pour $n - 1$ processus. Le deuxième montre que, pour $x < n-1$, il est impossible de construire un objet de consensus $(n, x + 1)$-vivace en utilisant des registres atomiques et des objets de consensus (n, x)-vivaces.

3.1.2.1 À propos des deuxièmes objets les plus puissants

Le résultat d'universalité de Herlihy implique que l'objet de consensus (n, n)-vivace est l'objet le plus puissant dans un système de n processus [69]. Gafni et Kuznetsov ont montré que, dans un système où la condition sans-attente est la seule condition de progression, l'objet de consensus sans-attente pour $n-1$ processus (ou l'objet de consensus $(n-1, n-1)$-vivace en utilisant notre notation) est le deuxième objet le plus fort dans un système de n processus [57]. Nos résultats montrent que, quand on considère les

conditions de progression asymétriques, l'objet de consensus $(n-1, n-1)$-vivace n'est *pas* le deuxième objet le plus puissant dans un système de n processus : l'objet de consensus $(n, n-2)$-vivace est plus fort que l'objet de consensus $(n-1, n-1)$-vivace (les objets de décision $(n, n-1)$-vivaces sont aussi (n, n)-vivaces : le processus pour lequel sans-attente n'est pas requis, s'il est correct, finira toujours par s'exécuter sans concurrence et terminera son exécution).

3.1.2.2 Définitions utilisées dans les démonstrations

Le modèle consiste en un ensemble de processus asynchrones qui communiquent en utilisant des objets partagés. Un *évènement* correspond à une instruction atomique exécutée par un processus. Par exemple, les évènements qui correspondent à un registre atomique sont de deux types : les évènements de lecture qui ne changent pas l'état du registre et les évènements d'écriture qui changent l'état du registre mais ne revoient pas de valeur. Nous utiliserons la notation e_p pour désigner un évènement arbitraire au processus p.

Exécution, implémentation, préfixe, extension Une *exécution* est une paire (f, S) où f est une fonction qui associe des états initiaux (valeurs) aux objets et S est une séquence d'évènements finie ou infinie. Une *implémentation* d'un objet utilisant un ensemble d'autres objets consiste en un ensemble non vide C d'exécutions, un ensemble Π de processus et un ensemble O d'objets partagés. Pour chaque évènement e_p au processus p, dans chaque exécution de C, l'objet accédé par e_p doit être dans O. Soient $x = (f, S)$ et $x' = (f', S')$ des exécutions. L'exécution x' est un *préfixe* de x (et x est une extension de x'), noté $x' \leq x$, si S' est un préfixe de S et $f = f'$. Quand $x' \leq x$, le suffixe de S obtenu en enlevant S' de S est noté $(x - x')$. Soit $S; S'$ la séquence obtenue en concaténant la séquence finie S et la séquence S'. La notation $x; S'$ sera alors utilisé comme l'abréviation de $(f, S; S')$.

Activé, indiscernable, déterministe Le processus p est *activé* dans l'exécution x s'il existe un évènement e_p tel que $x; e_p$ est une exécution. Pour simplifier, xp sera utilisé pour désigner $x; e_p$ si p est activé dans x, ou x dans le cas contraire. Le registre r est un registre *local* de p si p et seulement p peut accéder à r. Pour chaque séquence S, soit S_p la sous-séquence de S contenant tous les évènements de S au processus p. Les exécutions (f, S) et (f, S') sont *indiscernables* pour le processus p, noté $(f, S)[p](f', S')$, si $S_p = S'_p$ et $f(r) = f'(r)$ pour chaque registre local r de p. Les processus sont considérés comme déterministes, c'est-à-dire que si $x; e_p$ et $x; e'_p$ sont des exécutions, alors $e_p = e'_p$.

Valence, compatibilité, décideur Les démonstrations des théorèmes de cette section considèrent le cas du consensus binaire, c'est-à-dire que seules les valeurs 0 et 1 peuvent être proposées et décidées. Si la valeur v est proposée nous avons donc $v \in \{0, 1\}$. Soit $\overline{v} = 1 - v$. Elles utilisent également les notions suivantes. Une exécution finie x est *v-valente* si, dans toutes les extensions de x où une valeur est décidée, cette valeur est v ($v \in \{0, 1\}$). Une exécution est *univalente* si elle est soit 0-valente soit 1-valente, sinon elle est *bivalente*. Deux exécutions univalentes sont *compatibles* si elles ont la même valence, c'est-à-dire si elles sont toutes les deux soit 0-valentes, soit 1-valentes. Finalement, nous disons que le processus p est un décideur dans l'exécution x si, dans chaque extension y de x, l'exécution yp est univalente.

3.1.2.3 Démonstrations des résultats d'impossibilité

Lemme 3.1. *Pour chaque (implémentation d'un) objet de consensus sans-obstruction, si deux exécutions univalentes sont indiscernables pour un processus p et tous les objets accessibles par p sont dans le même état dans ces deux exécutions, alors ces exécutions sont compatibles.*

Démonstration. Soient w et y des exécutions univalentes telles que $w[p]y$ pour un processus p et l'état de tous les objets (locaux et partagés) accessibles par p est le même dans w et dans y. (Voir la figure 3.1(a).)

Considérons que w est v-valente. Par la définition de sans-obstruction, il y a alors une extension x de w constituée uniquement d'évènements de p où p décide v. L'exécution $z = y; (x - w)$ est aussi une exécution de l'algorithme telle que $z[p]x$. Comme p décide v dans z, z est v-valente. Or $y \leq z$ et y est univalente, y est donc v-valente. □

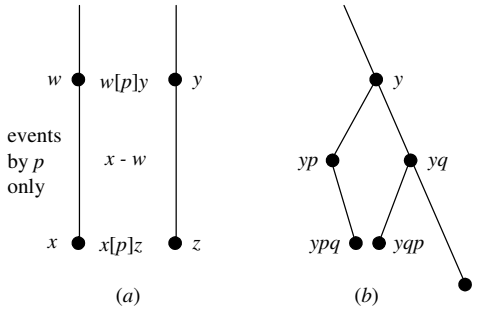

FIGURE 3.1 – Exécutions utilisées dans les démonstrations des lemmes 3.1 et 3.2

Lemme 3.2. *Soit y une exécution d'une implémentation d'un objet de consensus sans-obstruction et soient p et q deux processus différents tels que (1) $y \neq yp$ et $y \neq yq$, (2) les exécutions yp et yq sont univalentes et ne sont pas compatibles. Alors, durant leur premier évènement à partir de y, p et q accèdent au même objet et cet objet n'est pas un registre atomique.*

Démonstration. Considérons que durant le dernier évènement de yp, le processus p accède à un objet o et que durant le dernier évènement de yq, le processus q accède à un objet o'. (Voir la figure 3.1(b).)

Considérons d'abord le cas $o \neq o'$. Comme les deux évènements de p et de q sont indépendants, on a $ypq[q]yqp$ et l'état de tous les objets est le même dans ypq et dans yqp. Selon le lemme 3.1, ypq et yqp sont compatibles. Comme ypq est une extension de l'exécution univalente yp, yp et yqp doivent aussi être compatibles. C'est une contradiction avec l'hypothèse du lemme, on a donc $o = o'$.

Considérons maintenant que o est un registre atomique. En fonction de la dernière opération de p dans yp, il y a deux cas.

- Cas 1 : Dans yp, le dernier évènement est une écriture dans o par p. Comme p écrit dans o durant sa dernière opération à partir de y, la valeur de o doit être la même dans yp et dans yqp. (Nous utilisons le fait que l'écriture par p écrase tout changement éventuel de o par q.) On a donc $yp[p]yqp$ et l'état de tous les objets qui ne sont pas locaux à q est le même dans yp et dans yqp. Selon le lemme 3.1, yp et yqp sont donc compatibles, une contradiction.
- Cas 2 : Dans yp, le dernier évènement est une lecture de o par p. On a donc $yp[p]yqp$ et l'état de tous les objets qui ne sont pas locaux à p est le même dans ypq et dans yqp. Selon le lemme 3.1, ypq et yqp sont donc compatibles. Comme ypq est une extension de yp, yp et yqp doivent aussi être compatibles, une contradiction.

On a donc $o = o'$ et o n'est pas un registre atomique. □

Lemme 3.3. *Chaque objet de consensus sans-obstruction a une exécution vide bivalente.*

Démonstration. Par définition, l'exécution vide où toutes les entrées sont 0 doit être 0-valente, et de la même manière, l'exécution vide où toutes les entrées sont 1 doit être 1-valente. Soit p un processus arbitraire. Dans chaque exécution où toutes les entrées sont v, si p s'exécute seul, il finira par décider v. Selon le lemme 3.1, dans chaque exécution vide où l'entrée de p est $v \in \{0, 1\}$, si p s'exécute seul il doit finir par décider v. Cette observation implique que chaque exécution vide où toutes les entrées ne sont pas égales à 0 et où elles ne sont pas toutes égales à 1 doit être bivalente. □

Lemme 3.4. *Pour chaque objet de consensus $(n, 1)$-vivace, il existe une exécution bivalente x et un processus p tels que p est un décideur dans x.*

Démonstration. Soit $CONS$ un objet de consensus $(n, 1)$-vivace arbitraire qui est sans attente pour le processus p. Selon le lemme 3.3, $CONS$ a une exécution vide bivalente x_0. Nous commençons avec x_0 et continuons en respectant l'ordonnancement suivant qui préserve la bivalence :

$x \leftarrow x_0$; $done \leftarrow false$;
repeat
  if x has a bivalent extension yp
  /* extension which involves p */
    then $x \leftarrow yp$
    /* bivalent extension of x */
    else $done \leftarrow true$ **end if**
    /* no such bivalent extension */
until $done$ **end repeat**.

Comme $CONS$ est sans-attente pour p, la procédure précédente termine. Elle termine donc avec une exécution finie x telle que, pour toute extension y de x, l'exécution yp est univalente, et donc p est un décideur dans x. □

Lemme 3.5. *Chaque objet de consensus $(n, 1)$-vivace a une exécution bivalente y et deux processus p et q tels que : (1) p est un décideur dans y, (2) les exécutions yp et yqp sont univalentes et non compatibles, et (3) durant leur prochain évènement à partir de y, p et q accèdent au même objet et cet objet n'est pas un registre atomique.*

Démonstration. Soient $CONS$ un objet de consensus $(n, 1)$-vivace arbitraire et p le seul processus pour lequel $CONS$ garantit la condition sans-attente. Selon le lemme 3.4, il existe une exécution bivalente x telle que p est un décideur dans x.

Supposons que l'exécution xp soit v-valente. Comme x est bivalente, il existe une (plus petite) extension z de x qui est $\overline{v}$-valente. (Voir la figure 3.2(a).) Soit z' le plus long préfixe de z tel que $x[p]z'$. (Voir la figure 3.2(b).) Il y a deux cas possibles.

- Cas 1 : Si z' est univalent, on a $z' = z$. (Cela est dû au fait que (1) z' est le plus long préfixe de z tel que $x[p]z'$ et (2) z est la plus petite extension de x qui est $\overline{v}$-valente.) z' est donc $\overline{v}$-valente.
- Cas 2 : Si z' est bivalente, on a $z = z'p$. (Car $z - z'$ a un seul évènement et il est par p ; sinon z' ne serait pas le plus long préfixe de z tel que $x[p]z'$.) L'exécution $z'p$ est donc $\overline{v}$-valente.

Dans les deux cas, l'hypothèse que z est $\overline{v}$-valente implique donc que $z'p$ est $\overline{v}$-valente.

Considérons les extensions de x qui sont aussi des préfixes de z'. (Voir la figure 3.2(c).) Comme $x[p]z'$, pour chaque y telle que $x \leq y \leq z'$, on a $y \neq yp$. Comme xp et $z'p$ ne sont pas compatibles, il doit exister deux exécutions *différentes* y et yq telles que (1) $x \leq y < yq \leq z'$ et $p \neq q$ et (2) yp et yqp sont univalentes mais non compatibles. On obtient ensuite du lemme 3.2 que, durant leur prochain évènement à partir de y, p et q accèdent au même objet et cet objet n'est pas un registre atomique, ce qui conclut la démonstration du lemme. □

Lemme 3.6. *Chaque objet de consensus $(n, 1)$-vivace a une exécution bivalente y et deux processus p et q tels que : (1) p est un décideur dans y, (2) les exécutions yp et yqp sont univalentes et non compatibles et (3) durant leur prochain évènement à partir de y, tous les n processus accèdent au même objet et cet objet n'est pas un registre atomique.*

Démonstration. La preuve est par induction sur le nombre d'objets qui accèdent au même objet. La base de l'induction vient directement du lemme 3.5. On considère que le théorème est vrai pour $k < n$

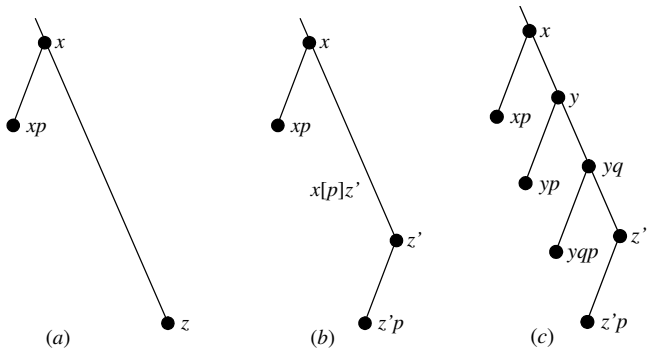

FIGURE 3.2 – Illustration des exécutions de la démonstration du lemme 3.5

processus et on le démontre pour $k+1$ processus.

Hypothèse d'induction. Chaque objet de consensus $(n,1)$-vivace a une exécution bivalente x et deux processus p et q tels que : (1) p est un décideur dans y, (2) les exécutions yp et yqp sont univalentes et non compatibles et (3) durant leur prochain évènement à partir de y, k des n processus accèdent au même objet o et cet objet n'est pas un registre atomique. On appelle K l'ensemble de ces processus et on considère que p et q sont dans k.

Étape d'induction. Soient x l'exécution mentionnée dans l'hypothèse d'induction et s un processus tel que $s \notin K$. Pour prouver que l'affirmation est vraie pour $k+1$ processus, on montre qu'il y a une extension y de x uniquement par des évènements de s telle que (1) p est un décideur dans y, (2) les exécutions yp et yqp sont univalentes et non compatibles et (3) durant leur prochain évènement à partir de y, les $k+1$ processus de $K \cup \{s\}$ accèdent au même objet o et cet objet n'est pas un registre atomique.

Supposons que l'exécution xp soit v-valente et l'exécution xqp $\overline{v}$-valente. (Voir la figure 3.3(a).) Comme x est bivalente, il existe une (plus petite) extension z de x uniquement par des évènements de s qui est univalente. Nous commençons par montrer que le processus s accède à o durant au moins un des évènements du suffixe $(z-x)$. Supposons, par contradiction, qu'aucun des évènements de $(z-x)$ impliquant s n'accède à o. Dans ce cas, on a (1) $x[p]z$ et l'état de tous les objets auxquels p peut accéder est le même dans x et dans z (car o est le seul objet auquel p accède dans x et dans z, et s n'accède pas à o dans z) et (2) $xq[p]zq$ et l'état de tous les objets auxquels p peut accéder est le même dans xq et dans zq (pour la même raison concernant z et parce que q est déterministe et donc accède à o avec la même opération dans xq et dans zq). Comme p est un décideur dans x (et est déterministe), (1) implique que xp et zp sont compatibles et (2) implique que xqp et zqp sont compatibles. Comme xp et xqp ne sont pas compatibles, zp et zqp ne sont pas non plus compatibles. Ce n'est pas possible car zp et zqp sont des extensions de l'exécution univalente z ; une contradiction. Le processus s accède donc à o durant au moins un des évènements du suffixe $(z-x)$.

Soit $y \geq x$ le plus long préfixe de z tel que aucun des évènements de $(y-x)$ n'accède à o. (Voir figure 3.3(b).) Comme $ys \leq z$, y est bivalent. De plus, durant son prochain évènement à partir de y, le processus s accède à o. Durant leurs prochains évènements à partir de y, les k processus de K accèdent aussi à o. On a donc (1) $x[p]y$ et l'état de tous les objets auxquels p peut accéder est le même dans x et dans y et (2) $xq[p]yq$ et l'état de tous les objets auxquels p peut accéder est le même dans xq et dans yq (Voir la figure 3.3(c).) Selon le même raisonnement que précédemment, xp et yp sont donc compatibles et xqp et yqp sont aussi compatibles. Comme xp et xqp ne sont pas compatibles, yp et yqp ne sont pas non plus compatibles. Finalement, comme p est un décideur dans x et $x \leq y$, p est donc un décideur dans y. □

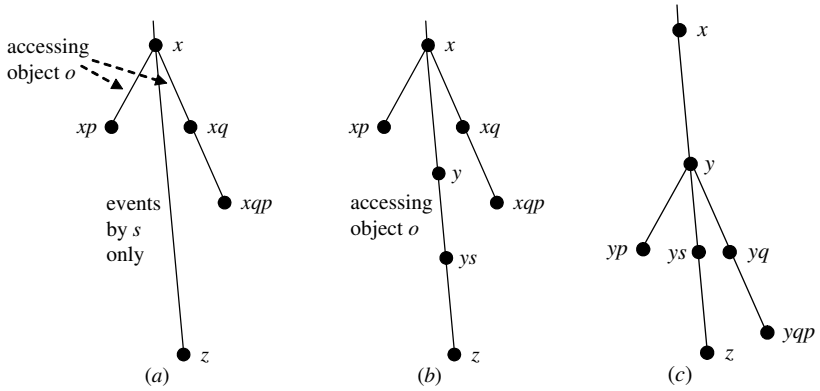

FIGURE 3.3 – Exécutions utilisées dans la démonstration du lemme 3.6

Théorème 3.7. *Il est impossible de construire un objet de consensus $(n, 1)$-vivace à partir d'objets de consensus $(n-1, n-1)$-vivaces et de registres atomiques.*

Démonstration. Selon le lemme 3.6, chaque implémentation d'un objet de consensus $(n, 1)$-vivace doit utiliser un objet o qui est accédé par tous les processus durant la même exécution et o n'est pas un registre atomique. Il est donc impossible de construire un objet de consensus $(n, 1)$-vivace en utilisant des objets de consensus $(n-1, n-1)$-vivaces (chacun ne pouvant être accédé que par $n-1$ processus au maximum) et des registres atomiques. □

Théorème 3.8. *Soit x un entier quelconque tel que $1 \leq x < n-1$. Il est impossible de construire un objet de consensus $(n, x+1)$-vivace en utilisant des objets de consensus (n, x)-vivaces et des registres atomiques.*

Démonstration. Supposons, par contradiction, qu'il existe une implémentation P d'un objet de consensus $(n, x+1)$-vivace utilisant uniquement des objets de consensus (n, x)-vivaces et des registres. Comme un objet de consensus $(n, x+1)$ vivace est aussi $(n, 1)$-vivace, selon le lemme 3.6, il existe une exécution y de P dans laquelle tous les processus accèdent au même objet o et cet objet n'est pas un registre atomique. Comme o n'est pas un registre, cet objet est donc un objet de consensus (n, x)-vivace.

Considérons maintenant qu'à la fin de y (juste avant que tous les processus accèdent à o), les x processus sans-attente crashent tandis que les $n-x$ autres processus accèdent à l'objet simultanément. Si $n - x > 1$, ces processus peuvent se trouver dans la situation où ils ne s'exécutent jamais seuls. La condition de progression de o ne garantit donc pas qu'un de ces processus reçoive une réponse de o. Mais la condition de progression de de l'implémentation P doit garantir qu'au moins un de ces processus reçoive une réponse de o ; une contradiction. □

3.1.3 La hierarchie de (n, x)-vivacité

Théorème 3.9. *Soit x un entier quelconque tel que $1 \leq x < n-1$. L'objet de consensus (n, x)-vivace a le nombre de consensus $(x+1)$.*

Démonstration. Montrons d'abord que le nombre de consensus d'un objet de consensus (n, x)-vivace est au moins $x+1$. Soient X l'ensemble prédéfini de x processus associé à l'objet et p un processus tel que $p \notin X$. Comme les processus de X sont sans-attente, il y a un temps fini après lequel aucun processus de X n'est en concurrence avec p ; p peut donc s'exécuter seul. Comme p est sans-obstruction, il termine aussi. Le nombre de consensus d'un objet de consensus $(x+1, x)$-vivace est donc au moins $x+1$. Étant donné un objet de consensus (n, x)-vivace, on peut le restreindre pour obtenir un objet de consensus

$(x+1,x)$-vivace. Un objet de consensus (n,x)-vivace a donc comme nombre de consensus au moins $x+1$.

Supposons maintenant, par contradiction, qu'un objet de consensus (n,x)-vivace a comme nombre de consensus au moins $x+2$. Selon la définition du nombre de consensus d'un objet, on peut donc construire un objet de consensus sans-attente dans un système de $x+2$ processus. Un tel objet satisfait de façon triviale la $(x+2,x+1)$-vivacité. Or, il a été montré dans le théorème 3.8 qu'un objet de consensus $(x+2,x+1)$-vivace ne peut pas être construit à partir d'objets de consensus $(x+2,x)$-vivaces et de registres atomiques. Un objet de consensus $(x+2,x)$-vivace ne peut donc pas avoir comme nombre de consensus $x+2$. Un tel objet a donc comme nombre de consensus exactement $x+1$. Ce résultat s'applique aussi aux objets de consensus (n,x)-vivaces avec $n > x+2$ (en empêchant les $n-(x+2)$ processus additionnels de participer), ce qui conclut la démonstration. □

Rappelons qu'un objet $(n,0)$-vivace garantit la terminaison sans-obstruction. La hierarchie de (n,x)-vivacité suivante est une conséquence du théorème 3.9 et du fait que les objets de consensus (n,n)-vivace et $(n,n-1)$-vivace ont tous les deux le nombre de consensus n. La notation $(n,x) \prec (n,y)$ signifie qu'il est possible de construire un objet de consensus (n,x)-vivace dans un système asynchrone en mémoire partagée enrichi avec des objets de consensus (n,y)-vivace alors que l'inverse est impossible. La notation $(n,x) \simeq (n,y)$ signifie qu'il est possible de construire un objet de consensus (n,x)-vivace dans un système asynchrone en mémoire partagée enrichi avec des objets de consensus (n,y)-vivace et vice-versa.

Corollaire 3.10. $(n,0) \prec (n,1) \prec \cdots \prec (n,x) \prec \cdots \prec (n,n-1) \simeq (n,n)$.

3.1.4 Conclusion

Cette section a introduit la notion de (y,x)-vivacité. Un objet est (y,x)-vivace si (1) il ne peut être accédé que par y des n processus qui constituent le système, (2) il garantit la condition sans-attente pour x des processus qui peuvent l'accéder et (3) il garantit la condition sans-obstruction pour les $y-x$ autres. Dans un système de n processus, (n,n)-vivace correspond à sans-attente et $(n,0)$-vivace correspond à sans-obstruction.

La (y,x)-vivacité lie la terminaison au motif de concurrence. La notion d'adversaire, définie dans [45], permet de lier la terminaison au motif de fautes. Un adversaire $\mathcal{A}$, défini pour un ensemble Π de processus, est un ensemble de sous-ensembles de Π. Un algorithme est $\mathcal{A}$-résilient si il garantit la terminaison pour chaque exécution dans laquelle l'ensemble S des processus qui crashent appartient à $\mathcal{A}$. Le lien entre (n,x)-vivacité et adversaire peut alors être fait de la façon suivante. Soit $\mathcal{A}$ l'ensemble constitué des sous-ensembles de Π qui ne contiennent pas tous les processus de X et de ceux qui contiennent tous les processus de Π sauf un. La (n,x)-vivacité correspond alors à la $\mathcal{A}$-résilience.

La section a ensuite apporté les contributions suivantes. Elle a d'abord montré qu'il est impossible de construire un objet de consensus $(n,1)$-vivace (c'est-à-dire un objet de consensus pour n processus qui est sans-attente pour l'un d'entre eux et sans-obstruction pour les autres) en utilisant des objets de consensus $(n-1,n-1)$-vivaces (c'est-à-dire des objets de consensus sans-attente pour $n-1$ processus) et des registres atomiques. Cela nous permet de mieux appréhender la frontière qui sépare les conditions sans-obstruction et sans-attente. Elle a ensuite montré que, dans un système de n processus, le nombre de consensus d'un objet de consensus (n,x)-vivace (avec $x < n$) est $x+1$. Cela établit une hiérarchie pour les objets de consensus (n,x)-vivaces.

3.2 x-sans attente

Dans la section précédente, nous avons présenté une famille de conditions de progression asymétriques fortes (quand on considère le problème du consensus, elles sont impossibles à obtenir dans un systèmes

de n processus même avec des objets de consensus pour $n-1$ processus). Nous allons maintenant présenter une famille de conditions de progression asymétriques nettement plus souples : la condition x-sans-attente.

Parmi les n processus du système, on distingue (comme précédemment) une partition du système en deux sous-ensembles de processus : un ensemble X de processus, avec $|X| = x$, pour lesquels on requiert la condition la plus forte et que nous appellerons *majeurs* ; les autres processus seront appelés *mineurs*. (Nous verrons que l'ensemble des majeurs peut être défini dynamiquement.)

De manière informelle, quand on considère le problème du consensus, la condition x-sans-attente est la suivante. Si (1) un majeur participe et n'est pas fautif ou (2) si aucun majeur ne participe et aucun mineur n'est fautif ou (3) un processus décide, tous les processus qui participent et doivent terminer. (Nous verrons que la notion de processus fautif, dans ce cas, est liée à une fenêtre de vulnérabilité.)

Nous présentons ensuite un algorithme qui construit un objet de consensus x-sans-attente dans un système de processus asynchrones qui peuvent crasher et qui communiquent en utilisant des registres atomiques et des objets de consensus sans-attente accessibles par seulement x processus. Comme nous le verrons, pour pouvoir implémenter le consensus x-sans-attente, l'algorithme doit résoudre un problème de compétition en présence de défaillances, ce qui n'est pas trivial. La construction universelle décrite dans [117] peut ensuite être utilisée pour construire des objets concurrents x-sans-attente en utilisant l'objet de consensus x-sans-attente pour n processus présenté ici.

Comparaison avec les conditions x-sans-obstruction et (n, x)-vivacité Dans un système de n processus, de manière similaire aux conditions n-sans-obstruction et (n, n)-vivacité, la condition n-sans-attente est en fait la condition sans-attente. Par contre, pour $x < n$, la condition x-sans-attente ne peut pas être comparée aux conditions x-sans-obstruction et (n, x)-vivacité. Les conditions x-sans-obstruction et (n, x)-vivacité dépendent du motif de concurrence ; quand plus de x processus s'exécutent en concurrence, la condition x-sans-obstruction ne garantit pas la terminaison. De même, quand plus d'un processus s'exécute, la condition (n, x)-vivacité ne garantit pas à $n - x$ processus du système qu'ils pourront terminer leur opération. Par contre, la condition x-sans-attente garantit la terminaison de tous les processus dès qu'un des majeurs corrects participe. L'hypothèse de x-sans-attente est donc souvent vérifiée dans les exécutions avec peu de crashes (ce qui arrive généralement en pratique).

3.2.1 Modèle de système utilisé

Processus asynchrones et modèle de crash Le système est un système classique de processus asynchrones communiquant par mémoire partagé comme défini dans le chapitre 2. Il est constitué de n processus asynchrones notés $p_1, \ldots, p_n$ Un processus exécute une séquence d'étapes définie par son algorithme. Il exécute son algorithme correctement jusqu'au moment où il crashe (s'il crashe). Après avoir crashé, un processus n'exécute plus aucune étape. Dans une exécution, un processus qui crashe est dit *fautif*. Sinon, il est dit *correct*.

Une partie de l'algorithme associé à un processus est appelée sa *fenêtre de vulnérabilité*. Un processus qui ne crashe pas durant son exécution de cette partie de son code est dit *bon*. Intuitivement, le crash d'un processus peut bloquer les autres seulement s'il n'est pas bon. Dans une exécution, un nombre arbitraire de processus peuvent crasher.

Modèle de communication Les processus communiquent en utilisant trois types d'objets.

- Des registres atomiques.
- Des objets snapshot (définis dans le chapitre 2).
  Ces objets peuvent être implémentés à partir de registres atomiques ; ils ne sont donc utilisés que pour faciliter l'écriture et la compréhension des algorithmes. Ils n'augmentent pas la puissance de calcul du système. Chacun de ces objets n'est écrit ici qu'une seule fois par les processus : étant

donné un objet snapshot $SM[1..n]$, le processus p_i écrit dans $SM[i]$ puis invoque SM.snapshot() (autant de fois que nécessaire). On obtient donc la propriété d'*inclusion* suivante. En supposant que SM est initialisé à $[\perp, \ldots, \perp]$, soient $snap1$ et $snap2$ deux invocations de SM.snapshot() qui renvoient $sm1$ et $sm2$ respectivement et qui sont telles que $snap1$ est placée avant $snap2$ dans l'ordre de linéarisation. On a alors $sm1 \leq sm2$ où $(sm1 \leq sm2) \equiv \big(\forall i : (sm1[i] \neq \perp) \Rightarrow (sm2[i] = sm1[i])\big)$.

– Des objets de consensus pour un ensemble de x processus où $x \geq 2$.
Un tel objet, qui ne peut être accédé que par un ensemble prédéfini X de $|X| = x$ processus, leur offre une seule opération (appelée ici xcons_propose() pour ne pas la confondre avec l'opération propose() de l'objet que l'on construira) qui permet à chaque processus de proposer une valeur et d'obtenir une valeur de décision. Ces objets sont sans-attente : toute invocation de l'objet par un processus correct termine. Rappelons que au plus une valeur peut être décidée et que cette valeur doit avoir été proposée.

À propos de la valeur de x On suppose que $x > 2$ mais il est intéressant de considérer le cas où $x = 1$ car il est légèrement différent des autres valeurs. Rappelons que $x = 1$ est le nombre de consensus le plus faible, celui des registres. Quand $x = 2$, on peut utiliser des objets Test&Set pour définir dynamiquement quels sont les processus qui constituent l'ensemble X (voir la section 3.2.3.4). Ce n'est plus possible quand $x = 1$. Cela signifie qu'un ensemble X de taille 1 ne peut pas être construit dynamiquement dans un système asynchrone où les processus communiquent seulement à travers des registres.

D'un autre coté, quand on considère le cas où l'ensemble X de taille 1 est défini statiquement, le système est doté d'un leader (éventuellement non fiable). Dans ce cas, dans toutes les exécutions où le leader ne crashe pas, le problème du consensus peut être résolu.

Cela montre une propriété intéressante qui différencie les systèmes dotés uniquement de registres des systèmes enrichis avec des objets de consensus pour x processus, où $x > 1$.

3.2.2 Consensus x-sans-attente pour n processus

Le but est de construire dans le système décrit précédemment un objet de consensus qui satisfait la condition de progression x-sans-attente. Cet objet offre aux processus l'opération xwf_decide().

Fenêtre de vulnérabilité La *fenêtre de vulnérabilité* d'un algorithme est définie dans [51] comme "un intervalle de temps durant l'exécution de l'algorithme dans lequel le délai ou l'inaccessibilité d'un seul processus peut forcer l'algorithme entier à attendre indéfiniment".

Tout en gardant son esprit, nous reformulons cette propriété comme suit : les fenêtre de vulnérabilité des algorithmes qui implémentent un objet sont les parties de leurs codes telles que le crash d'un processus durant une de ces parties peut provoquer le blocage permanent de processus corrects qui invoquent une opération sur cet objet [1].

Quelques prédicats Considérons un objet d'accord qui offre aux processus une opération decide(). Nous définissons les prédicats suivants.

– PART(i) est vrai si et seulement si p_i participe au consensus. D'un point de vue opérationnel, p_i participe à partir du premier accès à un objet partagé provoqué par decide().
– FAULTY(i) est vrai si et seulement si p_i crashe.
– GOOD(i) est vrai si et seulement si p_i ne crashe pas durant sa fenêtre de vulnérabilité.
– DEC(i) est vrai si et seulement si p_i termine son opération decide().

1. La notion de fenêtre de vulnérabilité ne considère plus un objet comme une "boite noire" mais comme une "boite blanche". Une approche similaire "boite noire/blanche" est parfois utilisée en ingénierie logicielle.

Si on ne prend pas en compte la notion de fenêtre de vulnérabilité, on peut considérer $\text{GOOD}(i) \equiv \neg\text{FAULTY}(i)$.

Consensus x-sans-attente Un objet de consensus x-sans-attente diffère d'un objet de consensus sans-attente classique dans sa propriété de terminaison. Plus précisément, il est défini par les propriétés suivantes où X et $\overline{X}$ désignent respectivement les ensembles de processus majeurs et mineurs.

- *Validité*. Une valeur décidée a été proposée par un processus.
- *Accord*. Tous les processus qui décident une valeur décident la même valeur.
- *Terminaison*. Si $P1 \vee P2 \vee P3$ (défini ci-dessous), tous les processus corrects qui participent terminent.
  - $P1 \equiv \big(\exists i \in X : \text{PART}(i) \wedge \text{GOOD}(i)\big)$,
  - $P2 \equiv \big(\forall i \in X : \neg\text{PART}(i)\big) \wedge \big(\forall i \in \overline{X} : \text{PART}(i) \Rightarrow \text{GOOD}(i)\big)$,
  - $P3 \equiv \big(\exists i : \text{DEC}(i)\big)$.

De manière informelle, la propriété de terminaison dit que tous les processus corrects qui participent décident si un processus majeur correct participe ou si aucun majeur ne participe mais tous les mineurs qui participent sont corrects. Il est important de remarquer que, pour chaque participant correct, cette spécification autorise des exécutions où sa valeur peut être décidée.

3.2.3 Construction d'un objet de consensus x-sans-attente pour n processus

Cette section présente une construction d'un objet de consensus x-sans-attente et démontre sa correction. Pour arriver à ce but, nous introduisons d'abord le type d'objet weak_agreement (une variante du type safe_agreement défini par Borowsky et Gafni [27], présenté dans la section 5.1.2.1) et son implémentation, que nous démontrons. L'objet de consensus x-sans-attente est ensuite construit de façon incrémentale. On considère d'abord le cas où l'ensemble X est défini statiquement. Finalement, la construction est enrichie pour le cas où X est défini dynamiquement.

3.2.3.1 Le type d'objet weak_agreement

Définition Le type d'objet weak_agreement offre deux opérations, qui sont notées $\text{wa\_propose}_i()$ et $\text{wa\_terminate}_i()$. Son but est de permettre aux processus, dans certaines conditions, de décider une seule valeur à partir des valeurs qu'ils proposent (quand ils invoquent l'opération $\text{wa\_propose}_i()$). Le but de $\text{wa\_terminate}_i()$ est de permettre aux processus d'indiquer que l'objet est devenu inutile (les processus bloqués dans l'objet, s'il y en a, sont alors débloqués). Comparé au type safe_agreement, le type weak_agreement permet de terminer dans plus de cas, le prix a payer étant de décider plusieurs valeurs différentes.

Plus formellement, le type weak_agreementest défini par les trois propriétés suivantes.

- *Validité*. Une valeur décidée a été proposée par un processus.
- *Accord*. Si aucun processus n'invoque $\text{wa\_terminate}_i()$, tous les processus qui décident une valeur décident la même valeur.
- *Terminaison*. Si (a) aucun processus ne crashe durant son exécution de $\text{wa\_propose}_i()$, ou (b) un processus décide une valeur (c'est-à-dire termine son invocation de $\text{wa\_decide}_i()$), ou (c) un processus termine son invocation de $\text{wa\_terminate}_i()$, chaque invocation de $\text{wa\_propose}_i()$ par un processus correct termine.

Construction d'un objet du type weak_agreement Cette construction est une modification simple de la construction du safe_agreement présenté dans [27] et dans la section 5.1.2.1. Elle utilise un registre atomique booléen $TERM$ (initialisé à $false$, passé à $true$ par l'opération $\text{wa\_terminate}_i()$) et deux objets snapshot $VAL[1..n]$ (initialisé à $[\bot, \ldots, \bot]$) et $PART[1..n]$ (initialisé à $[\emptyset, \ldots, \emptyset]$) qui sont utilisés par

```
init : for each j : 1 ≤ j ≤ n do VAL[j] ← ⊥ ; PART[j] ← ∅ end for ; TERM ← false.

operation wa_decide_i(v) :
(01)  VAL[i] ← v ;
(02)  val_i ← VAL.snapshot() ; participants_i ← {j | val_i[j] ≠ ⊥} ;
(03)  PART[i] ← participants_i ;
(04)  repeat part_i ← PART.snapshot() until
(05)     [∃j : (part_i[j] ≠ ∅) ∧ (∀k : [(k ∈ part_i[j]) ⇒ (part_i[k] ≠ ∅)])] ∨ TERM
(06)  end repeat ;
(07)  if (¬TERM)
(08)     then let s_part_i = smallest non-empty participant set in part_i[1..n] ;
(09)          let m = smallest process index in s_part ;
              % m is the smallest process index in the smallest participant set known by p_i%
(10)          val ← VAL.snapshot() ; let res = val[m]
(11)     else let res = v
(12)  end if ;
(13)  return(res).

operation wa_terminate_i() :
(14)  TERM ← true.
```

FIGURE 3.4 – Un implémentation du type weak_agreement (variante de [27])

les processus pour coopérer quand ils invoquent wa_propose$_i(v)$. $VAL[i]$ est utilisé par p_i pour déposer la valeur qu'il propose, tandis que $PART[i]$ est utilisé pour garder en mémoire l'ensemble des processus dont p_i observe la participation.

Les algorithmes qui implémentent wa_decide$_i()$ et wa_terminate$_i()$ sont décris dans la figure 3.4. Le comportement généré par wa_decide$_i()$ peut être décomposé en différentes phases.

- p_i commence par écrire la valeur v qu'il propose dans $VAL[i]$ (ligne 01). Ensuite, il lit de manière atomique l'ensemble des valeurs contenues dans le tableau $VAL[1..n]$ pour déterminer son observation de l'ensemble des participants. Ce sont les processus qui, du point de vue de p_i, ont écrit dans $VAL[1..n]$ (ligne 02). p_i écrit ensuite cet ensemble dans $PART[i]$ pour informer les autres processus de son observation de quels sont les processus qui participent (ligne 03).
  Observons que si un processus p_k crashe après avoir écrit dans $VAL[k]$ mais avant d'avoir écrit dans $PART[k]$, on a $VAL[k] \neq \bot \wedge PART[k] = \emptyset$ définitivement. Par contre, si p_k écrit dans $VAL[k]$ et crashe après avoir écrit dans $PART[k]$, on a définitivement $VAL[k] \neq \bot \wedge PART[k] \neq \emptyset$.
- p_i entre ensuite dans une boucle durant laquelle il lit de manière atomique le tableau $PART[1..n]$ jusqu'à ce qu'un prédicat soit satisfait. Ce prédicat est fait de deux sous-prédicats. Le plus simple, le booléen $TERM$, est satisfait quand un processus a invoqué wa_terminate$_i()$ (indiquant ainsi que la valeur calculée par cet objet weak_agreement est devenue inutile).
  La signification de l'autre sous-prédicat, c'est-à-dire

  $$\exists j : (part_i[j] \neq \emptyset) \wedge \big(\forall k : [(k \in part_i[j]) \Rightarrow (part_i[k] \neq \emptyset)]\big)$$

  est la suivante : il y a un processus p_j qui (a) a rendu publique son observation des participants ($part_i[j] \neq \emptyset$) et (b) chaque processus p_k perçu par p_j comme participant a également rendu publique son observation des participants ($part_i[k] \neq \emptyset$). Si (b) est satisfait, les processus de $PART[j]$ n'ont pas crashé entre leur écriture dans VAL et leur écriture dans $PART$. Il est important de remarquer que le prédicat utilisé à la ligne 05 est stable : quand il devient vrai, il reste vrai indéfiniment.
- Finalement, le processus p_i calcule la valeur qu'il décide pour cet objet weak_agreement. Si $TERM$ est vrai, p_i peut renvoyer n'importe quelle valeur (p_i décide donc la valeur qu'il propose). Si $TERM$ reste faux, tous les processus doivent décider la même valeur. Pour cela, p_i commence par déterminer le plus petit ensemble non vide de participants observés par un processus ($s\_part_i =$

$part_i[k]$), puis le plus petit index m parmi les participants de cet ensemble non vide. Finalement, il renvoie la valeur écrite dans $VAL[m]$. La démonstration montrera que, si $TERM$ reste faux, $VAL[m] \neq \perp$ et aucun processus ne décide une autre valeur.

Théorème 3.11. *L'algorithme présenté dans la figure 3.4 constitue une implémentation correcte du type* weak_agreement.

Démonstration. Validité. Après que le booléen ait été passé à vrai (si c'est le cas), un processus renvoie sa propre valeur et la validité est donc vérifiée trivialement. Considérons donc le cas où $TERM$ ne passe jamais à vrai. Nous devons montrer que le processus p_m (déterminé par p_i) a écrit précédemment dans $VAL[m]$. Selon le texte de l'algorithme, $VAL[m]$ contient soit $\perp$, soit la valeur proposée par p_m.

Comme le prédicat de la ligne 05 est vrai et stable (parce que $part_i$ n'est plus modifié), s_part_i existe. Cela signifie qu'il existe un processus p_j tel que $s\_part_i = PART[j]$. On a donc $m \in PART[j]$ et on en déduit que p_j observe p_m comme participant. On peut donc conclure que p_m a écrit dans $VAL[m]$ avant que p_j écrive dans $PART[j]$. On a donc $VAL[m] \neq \perp$ quand $VAL[m]$ est lu par p_i à la ligne 10, ce qui conclut la démonstration de la propriété de validité.

Terminaison. Observons d'abord qu'un processus ne peut rester bloqué que dans la boucle repeat (lignes 04-06). La démonstration consiste donc à montrer qu'un processus correct finit par sortir de cette boucle. Il y a trois cas à considérer.

- Un processus exécute $\mathsf{wa\_terminate}_i()$. Dans ce cas, aucun processus correct ne peut rester bloqué dans la boucle repeat.
- Aucun processus ne crashe pendant son exécution de $\mathsf{wa\_decide}_i()$. Dans ce cas, on finit par avoir $VAL[k] \neq \perp \wedge PART[k] \neq \emptyset$ pour chaque processus p_k qui invoque $\mathsf{wa\_decide}_k()$. Quand cela arrive, le prédicat de la ligne 05 devient vrai, ce qui prouve la terminaison dans ce cas.
- Un processus termine son invocation de $\mathsf{wa\_decide}_i()$. S'il termine parce que $TERM$ devient vrai, aucun processus ne peut rester bloqué dans la boucle. Considérons donc que $TERM$ reste faux.
  Chaque processus qui termine son invocation de $\mathsf{wa\_decide}_i()$ a d'abord effectué une dernière invocation de $PART.\mathsf{snapshot}()$ (ligne 04, cette invocation lui a permis de sortir de la boucle). Ces "dernières" invocations sont ordonnées par leur ordre de linéarisation. Soit p_i le processus dont la dernière invocation de $\mathsf{part\_snap1} = PART.\mathsf{snapshot}()$ est la première de ces "dernières" invocations.
  Soit p_ℓ ($\ell \neq i$) un processus correct qui invoque $\mathsf{wa\_decide}_\ell()$. Selon la définition de p_i le fait que p_ℓ est correct, il y a une invocation $\mathsf{part\_snap2} = PART.\mathsf{snapshot}()$ par p_ℓ après $\mathsf{part\_snap1}$. Selon la propriété d'inclusion de l'objet snapshot $PART$, on a $\mathsf{part\_snap1} \leq \mathsf{part\_snap2}$. L'entrée $part_i[j] = PART[j] \neq \emptyset$ qui a permis à p_i de sortir de la boucle permet donc aussi à p_ℓ de sortir de la boucle car on a alors $part_\ell[j] = PART[j]$, ce qui conclut la démonstration de la propriété de terminaison.

Accord. Si le booléen $TERM$ passe à vrai, la propriété d'accord est satisfaite trivialement. Considérons donc que $TERM$ reste à faux.

Selon la propriété d'inclusion de l'objet snapshot VAL, on a (a) tous les ensembles $participants_i$ sont calculés à la ligne 02, sont ordonnés par inclusion et donc, pour chaque paire d'ensembles $PART[j]$ et $PART[k]$ (écrits à la ligne 03), on a $PART[j] \subseteq PART[k]$ ou $PART[k] \subseteq PART[j]$ et (b) tous les ensembles de participants ayant la même taille contiennent les mêmes index de processus. De plus, comme pour chaque p_i on a (p_i a écrit $PART[i]$) $\Rightarrow$ ($i \in PART[i]$), les ensembles correspondants s_part_i définis à la ligne 08 sont bien définis (ils ne sont pas vides).

Chaque processus participant p_k tel que $k \notin PART[j]$ n'a pas encore écrit sa valeur proposée dans $VAL[k]$ (ligne 01) quand p_j invoque $VAL.\mathsf{snapshot}()$ (ligne 02), qui définit $participants_j$. Le processus

p_k calcule donc le snapshot qui définit $participants_k$ après p_j et donc, $PART[j] \subset PART[k]$. En prenant la contraposée de cette observation, les seuls processus p_ℓ qui peuvent satisfaire $PART[\ell] \subseteq PART[j]$ sont les processus tels que $\ell \in PART[j]$. Quand tous ces processus ont écrit leurs ensembles de participants $PART[\ell]$ (c'est-à-dire une fois que le prédicat à la ligne 05 est vérifié), le plus petit ensemble non vide de participants présent dans $PART[1..n]$ ne changera plus. Ceci est dû à l'observation suivante. $\forall \ell \in PART[k] \setminus PART[j]$ on a $PART[j] \subset PART[\ell]$, et donc $|last\_part_k[\ell]| = |PART[\ell]| \geq |PART[j]|$ (où $last\_part_k$ est la dernière valeur de $part_k$ lue par p_k quand il sort de la boucle), et donc $s\_part_k = s\_part_j$.

Tous les processus vont donc choisir le même plus petit ensemble et le même plus petit index parmi cet ensemble (ligne 09). Ils vont donc tous décider la même valeur, ce qui conclut la démonstration de la propriété d'accord. □

3.2.3.2 L'opération xwf_decide$_i()$: cas statique

Un processus détermine son statut majeur/mineur en invoquant l'opération major(i) qui renvoie vrai $true$ si et seulement si $i \in X$. On peut supposer que major() est implémenté par un démon du système. Dans le cas statique (dont on s'occupe dans cette section), le démon se base sur une affectation prédéfinie statiquement. Dans le cas dynamique (que nous considérerons ensuite), le démon utilise des objets Test&Set.

Un processus p_i qui veut participer au consensus invoque l'opération xwf_decide$_i(v)$ où v est la valeur qu'il propose. L'algorithme qui implémente cette opération est décrit dans la figure 3.5. Comme indiqué précédemment, il doit résoudre un problème de compétition. Il consiste à décider entre la valeur choisie par les majeurs et celle choisie par les mineurs.

Objets de base Les processus coopèrent en accédant aux objets partagés suivants.

- Un tableau à deux entrées $XCONS[0:1]$ d'objets de consensus pour x processus. Ces objets sont accédés uniquement par les objets de X (les majeurs).
  $XCONS[1]$ est utilisé par les x majeurs pour s'accorder sur une seule valeur parmi celles qu'ils proposent.
  $XCONS[0]$ est utilisé par les x majeurs pour s'accorder sur une seule valeur parmi celles proposées par les mineurs et connues des majeurs.
- Un objet weak_agreement WA. Cet objet est utilisé par les mineurs pour s'accorder sur une seule valeur parmi celles qu'ils proposent.
- Un tableau à deux entrées $PROP[0:1]$ initialisé à $[\perp, \perp]$.
  $PROP[1]$ contiendra la valeur décidée par les majeurs dans $XCONS[1]$.
  $PROP[0]$ contiendra la valeur décidée par les mineurs dans l'objet weak_agreement SA.
- $WINNER$ est un registre atomique, initialisé à $\perp$, qui finira par contenir une valeur parmi $\{0, 1\}$. Plus précisément, les majeurs et les mineurs sont en compétition pour imposer la valeur qui sera décidée. On aura $WINNER = 1$ si les majeurs gagnent cette compétition, et $WINNER = 0$ si les mineurs gagnent.

Comportement des processus Quand il invoque xwf_decide$_i()$, le processus p_i invoque d'abord major(i) (ligne 01). Ensuite, son comportement dépend du fait que p_i soit majeur ou mineur.

- Si c'est un majeur, p_i doit invoquer $XCONS[1]$.xcons_propose(v) pour que les majeurs s'accordent (entre eux) sur une seule valeur (ligne 02). Cette valeur est rendue publique en l'écrivant dans $PROP[1]$ (ligne 03).
  La valeur de $PROP[0]$ peut être $\perp$, une valeur proposée par un mineur, ou une valeur arbitraire (voir ci-dessous). Cela signifie que différents majeurs peuvent observer différentes valeurs de $PROP[0]$. Pour qu'ils s'accordent sur une seule valeur comme celle proposée par les mineurs, les

```
init : PROP[0 : 1] ← [⊥, ⊥] ; WINNER ← ⊥.

operation xwf_decide_i(v) :
(01) if (major(i))
(02)   then major_dec_i ← XCONS[1].xcons_propose(v) ;
(03)        PROP[1] ← major_dec_i ;
(04)        minor_prop_i ← XCONS[0].xcons_propose(PROP[0]) ;
(05)        if (minor_prop_i = ⊥) then WINNER ← 1 ; WA.wa_terminate_i()
(06)                              else WINNER ← 0 end if
(07)   else minor_dec_i ← WA.wa_decide(v) ;
(08)        PROP[0] ← minor_dec_i ;
(09)        major_prop_i ← PROP[1] ;
(10)        if (major_prop_i = ⊥) then WINNER ← 0 else wait( WINNER ≠ ⊥) end if
(11) end if ;
(12) let dec = PROP[WINNER] ;
(13) return(dec).
```

FIGURE 3.5 – Implémentation de l'opération xwf_decide$_i()$

majeurs utilisent le deuxième objet de consensus pour x processus $XCONS[0]$ (ligne 04). Si la valeur décidée dans $XCONS[0]$ est $\perp$, les majeurs gagnent la compétition avec les mineurs et donc p_i affecte 1 à $WINNER$ et invoque WA.wa_terminate$_i()$ pour signaler aux mineurs que l'objet WA n'est plus utile (ligne 05). Si les majeurs perdent cette compétition, p_i affecte 0 à $WINNER$ (ligne 06).

– Si c'est un mineur, p_i utilise d'abord l'objet weak_agreement WA (ligne 07). puis il dépose la valeur décidée par WA dans $PROP[0]$ pour la rendre publique (ligne 08). Ensuite, p_i lit la valeur (si elle existe) décidée par les majeurs entre eux (ligne 09). Si les majeurs ne se sont pas accordés sur une valeur, les mineurs déclarent unilatéralement qu'ils sont gagnants ; sinon, ils attendent de savoir quel groupe a gagné (ligne 10).

Finalement, p_i décide la valeur déterminée par les gagnants (lignes 12-13).

Fenêtres de vulnérabilité Un majeur qui crashe ne peut pas empêcher un autre majeur de décider ; il ne peut bloquer que des mineurs. Cela arrive quand les majeurs crashent après avoir écrit dans $PROP[1]$ et avant d'avoir affecté une valeur au registre atomique $WINNER$. Dans ce cas, un mineur correct peut être bloqué quand il exécute wait($WINNER \neq \perp$) (ligne 10). La fenêtre de vulnérabilité des majeurs est donc faite des lignes 03-06.

Un mineur qui crashe ne peut pas bloquer indéfiniment un majeur correct. Par contre, un mineur qui crashe entre la ligne 01 et la ligne 03 de l'opération WA.wa_decide() peut bloquer d'autres mineurs quand ils exécutent la boucle dans WA.wa_decide(). La fenêtre de vulnérabilité des majeurs est donc la ligne 02 de WA.wa_propose() (figure 3.4) qui est invoquée à la ligne 07 de l'opération xwf_decide$_i()$ (figure 3.5).

3.2.3.3 Démonstration de la construction

Théorème 3.12. *L'algorithme décrit dans la figure 3.5 est une implémentation correcte du type d'objet consensus x-sans-attente.*

Démonstration. Validité La valeur décidée est soit la valeur présente dans $PROP[1]$ (une valeur proposée par un majeur), soit la valeur présente dans $PROP[0]$ (une valeur proposée dans un mineur).

Si la valeur décidée est celle présente dans $PROP[1]$, alors $WINNER$ a reçu la valeur 1 d'un majeur avant qu'aucun processus ne décide. Dans ce cas, un majeur a écrit le résultat de l'opération $XCONS[1]$.xcons_propose() dans $PROP[1]$ avant d'écrire dans $WINNER$ et cette valeur a été proposée par un majeur. La valeur décidée a alors été proposée par un majeur.

Si la valeur décidée est celle présente dans $PROP[0]$, alors $WINNER$ a reçu la valeur 0 soit d'un majeur, soit d'un mineur avant qu'aucun processus ne décide. Si $WINNER$ a reçu la valeur 0 d'un majeur, un des majeurs a observé $PROP[0] \neq \perp$ (lignes 04-06). Cette valeur a été écrite dans $PROP[0]$ par un mineur et est la valeur renvoyée par WA.wa_decide() à ce mineur. La valeur présente dans $PROP[0]$ a donc été proposée par un mineur. Si $WINNER$ a reçu la valeur 0 d'un mineur, ce processus a écrit dans $PROP[0]$ la valeur renvoyée par WA.wa_decide(). La valeur décidée a alors été proposée par un des mineurs, ce qui conclut la démonstration de la propriété de validité.

Accord La démonstration repose sur (a) le fait qu'un majeur (respectivement mineur) écrit d'abord dans $PROP[1]$ (respectivement $PROP[0]$) et ensuite lit $PROP[0]$ (respectivement $PROP[1]$) et (b) l'atomicité des registres qui implique que si $PROP[1]$ est écrit avant $PROP[0]$, tous les mineurs liront $PROP[1] \neq \perp$, et si $PROP[0]$ est écrit avant $PROP[1]$, tous les majeurs liront $PROP[0] \neq \perp$.

Si $WINNER$ a reçu 1 d'un des majeurs (ligne 05), alors un des majeurs (pas nécessairement le même) a observé $PROP[0] = \perp$ et a proposé $\perp$ dans $XCONS[0]$. Tous les autres majeurs qui exécutent la ligne 05 écriront aussi 1 dans $WINNER$. Avant d'invoquer l'opération $XCONS[0]$.xcons_propose(), le processus qui a observé $PROP[0] = \perp$ a écrit la valeur renvoyée par $XCONS[1]$.xcons_propose() dans $PROP[1]$. Tous les mineurs observeront donc $PROP[1] \neq \perp$ et n'écriront pas dans $WINNER$ (ligne 10). La valeur de $WINNER$ ne changera pas de 1 vers 0.

Si $WINNER$ a reçu la valeur 0, alors un des majeurs a observé $PROP[0] \neq \perp$ et l'a proposé à $XCONS[0]$ ou un mineur a observé $PROP[1] = \perp$. Si un mineur a observé $PROP[1] = \perp$, il a écrit la valeur renvoyée par WA.wa_propose() dans $PROP[0]$ avant de lire $\perp$ dans $PROP[1]$. Tous les majeurs observeront donc $PROP[0] \neq \perp$ et $\perp$ ne sera pas renvoyé par $XCONS[0]$.xcons_propose(). Tous les majeurs écriront alors 0 dans $WINNER$. Parce qu'un mineur ne peut pas écrire 1 dans $WINNER$ la valeur de $WINNER$ ne changera pas de 0 vers 1. Une fois que $WINNER$ a reçu une valeur, elle ne changera donc pas.

Si $WINNER = 1$, la valeur décidée est la valeur présente dans $PROP[1]$. C'est la valeur renvoyée par $XCONS[1]$.xcons_propose() aux majeurs. Parce que cette valeur est unique et que seuls les majeurs peuvent écrire dans $PROP[1]$, la même valeur est décidée par tous les processus.

Si $WINNER = 0$, la valeur décidée est la valeur présente dans $PROP[0]$. C'est la valeur renvoyée par WA.wa_propose() aux mineurs. Parce qu'un majeur qui n'écrit pas 1 dans $WINNER$ n'exécute pas WA.wa_terminate() et à cause des propriétés du type weak_agreement, cette valeur est unique. Parce que seuls les mineurs écrivent dans $PROP[0]$, la même valeur est décidée par tous les processus, ce qui conclut la démonstration de la propriété d'accord.

Terminaison. Rappelons qu'un *bon* processus est un processus qui ne crashe pas dans sa fenêtre de vulnérabilité.

Un majeur n'exécute jamais de boucle ou d'attente et donc tous les majeurs corrects qui participent terminent. Un mineur peut être bloqué durant l'exécution de WA.wa_propose() ou pendant l'instruction d'attente à la ligne 10 si $WINNER = \perp$.

Si un bon majeur participe et écrit 1 dans $WINNER$, il invoquera WA.wa_terminate() et donc, à cause des propriétés du type weak_agreement, tous les mineurs corrects qui participent termineront. Si un bon majeur participe et écrit 0 dans $WINNER$, un des majeurs a observé $PROP[0] \neq \perp$, ce qui signifie qu'un mineur a terminé son invocation de WA.wa_decide() et a écrit dans $PROP[0]$. Encore à cause des propriétés du type weak_agreement, tous les mineurs corrects qui participent termineront.

Si aucun majeur ne participe et si tous les mineurs sont bons, les propriétés du type weak_agreement garantissent que tous les mineurs corrects terminent leur invocation de WA.wa_decide(). Ils observeront alors $PROP[1] = \perp$ et n'exécuteront pas l'instruction d'attente. Ils termineront donc tous.

Nous avons déjà montré que si un majeur décide, tous les processus corrects décident. Si un mineur p_i décide, alors il a terminé son invocation de SA.wa_decide() et il a soit écrit 0 dans $WINNER$, soit

```
operation major(i) :
(01) ℓ ← 1 ; major ← false ;
(02) while (ℓ ≤ x ∧ ¬major) do
(03)    major ← TS[ℓ].test&set() ; ℓ ← ℓ + 1
(04) end while ;
(05) return(major).
```

FIGURE 3.6 – Implémentation de l'opération major()

observé $WINNER = 1$ (sinon, il n'aurait pas décidé). Comme p_i décide, selon les propriétés du type weak_agreement, chaque mineur correct p_j terminera son invocation de WA.wa_decide(). Comme on finira par avoir $WINNER \neq \perp$, p_j décidera, ce qui conclut la démonstration de la propriété de terminaison. □

3.2.3.4 L'opération xwf_decide$_i$() : cas dynamique

Cette section considère le cas où, en supposant $x \geq 2$, l'ensemble X est défini dynamiquement. Intuitivement, les majeurs sont les x premiers processus qui invoquent l'opération xwf_decide$_i$() (figure 3.5). Observons que, comme X est défini statiquement, les objets de consensus pour x processus $XCONS[0]$ et $XCONS[1]$ utilisés par cette opération ne sont plus connus statiquement. On doit donc remplacer l'invocation des opérations $XCONS[a]$.xcons_propose() aux lignes 02 et 04 de xwf_decide$_i$() par des invocations d'une opération prenant en compte l'aspect dynamique et définie de manière appropriée.

L'opération major(i) Cette opération définit l'ensemble des majeurs comme les x "premiers" processus qui l'invoquent. (Remarquons que n'importe quelle opération qui renvoie vrai à au plus x processus pourrait être utilisée.) La notion de "premiers" est définie grâce à un tableau d'objets Test&Set de taille x noté $TS[1..n]$.

Un tel objet a une seule opération, notée test&set(), qui renvoie vrai uniquement au premier processus qui l'invoque. Comme les objets Test&Set sont linéarisables [77], "premier" est bien défini. Finalement, remarquons qu'il est possible de construire un objet Test&Set qui peut être accédé par n'importe quel nombre de processus à partir d'un objet Test&Set qui ne peut être accédé que par deux processus prédéterminés [63]. Un objet Test&Set peut donc être implémenté avec le consensus pour 2 processus, ce qui correspond à l'hypothèse $x \geq 2$.

L'algorithme qui implémente major() est décrit dans la figure 3.6. Elle renvoie $true$ aux x premiers processus qui l'invoquent, tandis qu'elle renvoie $false$ aux suivants. L'ordre de linéarisation est défini comme suit. Une opération major() qui renvoie $true$ est linéarisé à l'instant de l'invocation $TS[\ell]$.test&set() qui lui renvoie $true$ et une opération major() qui renvoie $false$ est linéarisé à l'instant de son invocation $TS[x]$.test&set().

L'opération dyn_xcons_propose$_i$() Pour résoudre le problème causé par le fait que les x majeurs sont définis dynamiquement, deux tableaux d'ensembles et d'objets de consensus pour x processus sont utilisés. Soit m le nombre de sous-ensembles de taille x dans l'ensemble des $n \geq x$ index de processus. On a :

- $SET\_LIST[1..m]$ est un tableau contenant les m sous-ensembles de taille x. $SET\,LIST[\ell]$ contient le sous-ensemble identifié par ℓ.
- $XCONS[1..m]$ est un tableau de m objets de consensus pour x processus. $XCONS[\ell]$ est l'objet de consensus qui ne peut être accédé que par les x processus dont les index définissent l'ensemble $SET\,LIST[\ell]$.

L'opération dyn_xcons_propose$_i$() est décrite dans la figure 3.7. Un majeur p_i commence par scanner $SET\_LIST[1..m]$ (tous les majeurs scannent cette liste dans le même ordre). Quand il rencontre un

```
operation dyn_xcons_propose_i(v) :
(01) res ← v ;
(02) for ℓ from 1 to m do
(03)   if (i ∈ SET_LIST[ℓ]) then res ← XCONS[ℓ].xcons_propose(res) end if ;
(04) end for ;
(05) return(res).
```

FIGURE 3.7 – Implémentation de l'opération dyn_xcons_propose$_i()$

ensemble $SET\_LIST[\ell]$ tel que $i \in SET\_LIST[\ell]$, p_i invoque $XCONS[\ell]$.dyn_xcons_propose$_i(res)$ et adopte la valeur renvoyée comme son estimation actuelle de la valeur décidée. On peut remarquer que, quelque soit l'ensemble X, comme tous les majeurs qui participent et qui ne crashent pas scannent toutes la liste, il invoque nécessairement $XCONS[\ell]$.dyn_xcons_propose$_i()$ où ℓ est tel que $SET\_LIST[\ell] = X$.

Redéfinition du code des lignes 02 et 04 de xwf_decide$_i()$ Le tableau $XCONS[0 : 1]$ d'objets de consensus pour x processus est remplacé par le tableau $DXCONS[0 : 1]$ dans lequel chaque entrée contient un seul objet dont la seule opération est dyn_xcons_propose$_i()$.

Il est important de remarquer que, alors que les implémentations de $DXCONS[0]$ et de $DXCONS[1]$ peuvent partager le même tableau $SET\_LIST[1..m]$, chacune d'elle doit utiliser son propre tableau d'objets de consensus pour x processus, respectivement $XCONS[0][1..m]$ et $XCONS[1][1..m]$.

Chapitre 4

Abstractions

Pour pouvoir écrire des programmes corrects, il est nécessaire de pouvoir comprendre leur fonctionnement en détail. Une technique utilisée pour faciliter la compréhension des programmes depuis les débuts de l'informatique est la factorisation de certaines parties du code d'un programme ; c'est l'appel de fonction. Cela permet au programmeur de masquer la complexité d'une partie de son code : son écriture, sa lecture et sa correction sont alors beaucoup plus faciles.

Registres atomiques dans un système hybride Sur le même principe, nous verrons dans ce chapitre des abstractions permettant de faciliter la conception d'algorithmes distribués. La première est celle sur laquelle se base tous les autres algorithmes de cette thèse : la mémoire partagée, c'est-à-dire les registres atomiques. Nous introduisons d'abord un nouveau modèle de système hybride où les processus sont partitionnés en groupes : ils ont accès à des registres partagés parmi les membres de leur groupe, mais deux processus membres de groupes différents ne peuvent communiquer qu'en échangeant des messages. Dans un tel système, relativement complexe, il est important de pouvoir fournir au programmeur des outils simples permettant la communication entre les différents processus. Nous verrons qu'il est impossible de construire des registres atomiques de manière sans-attente dans un tel système. Nous verrons ensuite quelle est l'aide minimale à apporter pour permettre la construction de registres atomiques dans un tel système (sous la forme d'un détecteur de fautes). Cette aide est plus faible que celle nécessaire pour construire des registres atomiques dans un système à passage de messages classiques. Le système hybride présenté ici est donc plus faible qu'un système à mémoire partagée classique, mais plus fort qu'un système à passage de message classique.

Snapshot et snapshot partiel Nous aborderons ensuite l'abstraction du *snapshot*. Cette abstraction, présentée dans le chapitre 2, consiste en un ensemble de registres partagés par les processus. Les processus peuvent écrire dans ces registres, comme dans les registres atomiques classiques ; la différence se situe au niveau de l'opération de lecture. Avec des registres atomiques classiques, un processus qui veut lire plusieurs registres doit les lire les uns après les autres, de manière asynchrone. Rien ne garantit donc que certains de ces registres ne soient pas modifiés durant cette lecture. Avec le snapshot, les processus peuvent lire l'ensemble des registres de manière atomique : la lecture de tous ces registres apparait au processus qui l'exécute comme ayant eu lieu instantanément. Il est intéressant de remarquer que cette abstraction n'augmente pas la puissance du système : on peut implémenter les opérations du snapshot sans-attente dans un système asynchrone en utilisant uniquement des registres atomiques.

Pour arriver à implémenter ces opérations de manière sans-attente, les algorithmes de snapshot doivent intégrer à leur opération d'écriture la lecture de l'ensemble des registres. Pour cette opération, les algorithmes classiques commencent par lire, puis écrivent. Nous verrons un nouvel algorithme qui inverse cet ordre : pendant leur opération d'écriture, les processus écrivent d'abord leur valeur en mémoire partagée, la rendant publique, puis calculent la valeur de l'ensemble des registres.

Dans certains cas, les processus n'ont pas besoin de lire tous les registres du système de manière atomique ; ils n'ont besoin que d'un sous-ensemble. L'algorithme présenté ici est un algorithme de snapshot *partiel*, c'est-à-dire un algorithme qui ne renvoie que les valeurs dont les processus ont besoin. On peut obtenir ce résultat trivialement avec un algorithme de snapshot classique, mais pas de manière efficace. La complexité de l'algorithme présenté ici dépend du nombre de registres lus par le processus ou les processus concurrents et permet donc d'obtenir une complexité indépendante du nombre total de registres dans le système.

4.1 Construction d'une mémoire partagée dans un système hybride

Vers de nouveau modèles de systèmes La multiplication des cœurs dans les processeurs a créé de nouveau défis pour les concepteurs d'architectures. En effet, en présence de centaines, voire de milliers de cœurs, l'utilisation d'une mémoire partagée entre tous les cœurs entraine une perte de performance : le bus mémoire est engorgé. Les constructeurs de processeurs envisagent donc de nouvelles pistes, par exemple des processeurs dans lesquels les différents cœurs du même processeur ne communiquent que par passage de message. De plus, dans les fermes de serveurs nécessaires au cloud-computing, les serveurs utilisent déjà des processeurs multi-cœurs à mémoire partagée, mais la communication entre différents serveurs ne peut se faire qu'en échangeant des messages.

Dans cette section, nous introduisons un nouveau modèle de système hybride, où les processus sont partitionnés en groupes. Chaque groupe a accès à une mémoire partagée, mais cette mémoire ne peut être accédée que par les membres du groupe. Deux processus de groupes différents ne peuvent communiquer que par passage de message. Ce modèle est noté $SM\_MP_{n,m}[\emptyset]$ (où n est le nombre de processus du système et m est le nombre de groupes). Un tel système ne permet pas, sans aide additionnelle, de construire un registre atomique partagé par tous les processus du système, ce qui est prouvé par une démonstration formelle.

Nous introduisons ensuite un nouveau détecteur de fautes $M\Sigma$. Ce détecteur de fautes permet d'implémenter un registre atomique dans le système précédent ; nous le montrons grâce à un algorithme, dont nous démontrons la correction. $M\Sigma$ est aussi le plus faible détecteur de fautes pour arriver à ce but, ce qui est démontré en obtenant $M\Sigma$ à partir de n'importe quel algorithme basé sur un détecteur de fautes permettant l'implémentation d'un registre atomique dans ce système.

Finalement, nous montrons que $M\Sigma$ est strictement plus faible que Σ, le plus faible détecteur de fautes pour implémenter un registre atomique dans un système à passage de message classique. Le nouveau modèle hybride présenté ici est donc plus faible qu'un système à mémoire partagé, mais plus fort qu'un système à passage de message. Nous présentons aussi une condition nécessaire et suffisante pour implémenter $M\Sigma$ dans un système hybride.

4.1.1 Un modèle de système à communication hybride

Processus asynchrones et modèle de crash Le système est un système classique de processus asynchrones communiquant par mémoire partagé comme défini dans le chapitre 2. Il est constitué de n processus asynchrones notés $p_1, \ldots, p_n$ Un processus exécute une séquence d'étapes définie par son algorithme. Il exécute son algorithme correctement jusqu'au moment où il crashe (s'il crashe). Après avoir crashé, un processus n'exécute plus aucune étape. Dans une exécution, un processus qui crashe est dit *fautif*. Sinon, il est dit *correct*.

Condition de progression nous considérons ici uniquement la condition sans-attente, c'est-à-dire que tous les processus corrects doivent terminer toutes leurs opérations (dans le cas d'un registre REG, REG.read() et REG.write()) dans un temps fini, indépendamment de la concurrence et des crashs des

autres processus. On considère donc que la concurrence n'est pas bornée et qu'il peut y avoir jusqu'à $t = n - 1$ crashs dans le système.

Communication par passage de message Les processus peuvent envoyer et recevoir des messages à travers des canaux de communication fiables. On considère que chaque paire de processus est reliée par un canal bidirectionnel. Les canaux sont fiables mais asynchrones. Fiable signifie que les messages ne peuvent pas être corrompus, ni dupliqués ou perdus. Asynchrone signifie que le délai d'arrivée d'un message est fini mais arbitraire.

L'envoi et la réception d'un message sont des opérations atomiques. Les processus peuvent aussi utiliser une opération de broadcast (qui envoie le message à tous les processus), mais cette opération n'est pas atomique (si un processus crashe pendant une opération de broadcast, un sous-ensemble arbitraire des processus reçoit le message).

Communication par une mémoire partiellement partagée Les n processus sont partitionnés en m, $1 \leq m \leq n$, ensembles non vides $P[1], \ldots, P[m]$ appelés clusters (c'est-à-dire $\cup_{1 \leq x \leq m} P[x] = \Pi$ et $\forall x, y : (x \neq y) \Rightarrow (P[x] \cap P[y] = \emptyset)$).

Dans chaque cluster x, $1 \leq x \leq m$, les processus de $P[x]$ partagent une mémoire en lecture/écriture notée MEM_x. MEM_x est composée de registres atomiques 1RMW (*single-writer/multi-reader*, mono-écrivain/multi-lecteur) [1]. Les registres sont désignés de la façon suivante : si $i \in P[x]$, $MEM_x[i]$ ne peut être écrit que par p_i et peut être lu par tous les processus de $P[x]$ (si $i \notin P[x]$, $MEM_x[i]$ n'existe pas car p_i ne peut pas accéder à MEM_x).

Deux exemples de mémoire partiellement partagée sont présentés dans la figure 4.1 (les canaux de communication ne sont pas représentés). Dans les deux cas, $n = 7$ et $m = 3$ mais les partitions sont différentes.

FIGURE 4.1 – Deux exemples de mémoire partiellement partagée

Notation Comme indiqué précédemment, la notation $SM\_MP_{n,m}[\emptyset]$ est utilisée pour désigner le modèle hybride de base. Dans la suite, $\emptyset$ sera remplacé par un détecteur de fautes pour désigner le modèle enrichi correspondant. Dans la figure 4.1, nous avons deux instances de $SM\_MP_{7,3}[\emptyset]$.

Deux cas particuliers Les deux cas extrêmes $m = 1$ et $m = n$ sont particulièrement intéressants. Dans le cas $m = 1$, tous les processus ont accès à la même mémoire partagée. Le modèle en mémoire partagée étant plus puissant que le modèle en passage de message, la communication par messages devient inutile. $SM\_MP_{n,1}[\emptyset]$ correspond donc au cas d'un système à mémoire partagée classique.

Quand $m = n$, chaque processus est seul dans sont groupe. Pour chaque x, $1 \leq x \leq n$, MEM_x est donc locale au processus qui peut l'accéder. $SM\_MP_{n,n}[\emptyset]$ correspond donc au cas d'un système à passage de message classique.

1. On peut construire des registres atomiques classiques multi-écrivain/multi-lecteur à partir de registres mono-écrivain/multi-lecteur [25, 99, 102] et vice-versa, cette hypothèse n'entraine donc pas de perte de généralité.

4.1.1.1 Un registre atomique ne peut pas être construit dans $SM\_MP_{n,m}[\emptyset]$ quand $m > 1$

Théorème 4.1. *Soit $1 < m \leq n$. Il est impossible de construire un registre atomique sans-attente dans $SM\_MP_{n,m}[\emptyset]$.*

Démonstration. La preuve consiste en une réduction simple vers la preuve d'impossibilité de l'implémentation sans-attente d'un registre dans un système asynchrone à passage de message [16, 25, 102, 110].

Pour cela, considérons qu'il existe un algorithme A qui construit, de manière sans-attente, un registre dans $SM\_MP_{n,m}[\emptyset]$ et considérons ses exécutions dans $SM\_MP_{n,m}[\emptyset]$ où, dans chaque partition, tous les processus crashent avant leur première instruction sauf un. Comme A est sans-attente, les $m > 1$ processus restants construisent un registre dans le modèle $SM\_MP_{m,m}[\emptyset]$, c'est-à-dire dans un système à passage de message classique. Cela contredit l'existence de A, ce qui conclut la preuve. □

4.1.2 Une nouvelle classe de détecteurs de fautes

4.1.2.1 Motifs de fautes et détecteurs de fautes

Le *modèle temporel* est l'ensemble $\mathbb{N}$ des entiers naturels. Cette notion de temps n'est pas accessible aux processus. Elle ne peut être utilisée que d'un point de vue extérieur, pour définir ou démontrer des propriétés. Les instants sont notés τ, τ', etc.

Définitions formelles Les notions introduites ici viennent de [36].

Un *motif de fautes* est une fonction $F()$ telle que $F(\tau)$ dénote l'ensemble des processus qui ont crashé à l'instant τ. Comme les crashs sont stables, on a $\forall\ \tau : F(\tau) \subseteq F(\tau + 1)$. Étant donnée une exécution, soit $\mathcal{F}$ l'ensemble des processus qui crashent durant cette exécution (les processus fautifs) et $\mathcal{C}$ l'ensemble des processus qui ne crashent pas (les corrects). On a $\mathcal{F} = \cup_\tau F(\tau)$ et $\mathcal{C} = \Pi \setminus \mathcal{F}$.

Une *histoire de détecteur de fautes H* d'image $\mathcal{R}$ est une fonction de $\Pi \times \mathbb{N}$ vers $\mathcal{R}$ qui peut être interprétée de la façon suivante : $H(i, \tau)$ est la sortie du détecteur de fautes considéré au processus p_i à l'instant τ.

Un *détecteur de fautes $\mathcal{D}$ d'image $\mathcal{R}$* est une fonction qui associe chaque motif de fautes $F()$ à un ensemble non vide d'histoires de détecteur de fautes d'image $\mathcal{R}$. $\mathcal{D}(F)$ est l'ensemble des comportements (histoires de détecteurs de fautes) que $\mathcal{D}$ peut avoir quand le motif de fautes est F.

Du point de vue opérationnel Du point de vue d'un algorithme, un détecteur de fautes peut être vu comme un mécanisme distribué qui fournit à chaque processus p_i une variable locale en lecture seule dont la valeur à l'instant τ est $H(i, \tau)$.

4.1.2.2 La classe de détecteurs de fautes Σ

Comme il a été indiqué précédemment, cette classe de détecteurs de fautes [43] est la classe des plus faibles détecteurs de fautes qui permettent l'implémentation d'un registre dans un système à passage de message. En utilisant le formalisme introduit précédemment, cela signifie que $SM\_MP_{n,n}[\Sigma]$ est le modèle de système basé sur un détecteur de fautes le plus faible dans lequel un registre peut être construit.

L'image de Σ est l'ensemble de tous les sous-ensembles non vides de processus ($2^\Pi \setminus \emptyset$). Soit Σ_i la variable locale en lecture seule que Σ fournit à p_i. Une telle sortie locale est appelée un *quorum*. Cette classe de détecteurs de fautes est définie par les deux propriétés suivantes dans lesquelles Σ_i^τ désigne la valeur de Σ_i à l'instant τ, c'est-à-dire $\Sigma_i^\tau = H(i, \tau)$.

- *Intersection*. $\forall\ i, j \in \Pi,\ \forall\ \tau, \tau' :\ \Sigma_i^\tau \cap \Sigma_j^{\tau'} \neq \emptyset$.
- *Vivacité*. $\exists\ \tau :\ \forall\ \tau' \geq \tau :\ \forall\ \in \mathcal{C} :\ \Sigma_i^\tau \subseteq \mathcal{C}$.

La propriété d'intersection établit que toute paire quelconque de quorums pris à un instant quelconque s'intersecte. Cette propriété empêche le partitionnement et est utilisée pour maintenir la cohérence du registre atomique. La propriété de vivacité établit qu'après un temps fini, les processus fournis par un quorum sont tous corrects. Cette propriété est utilisée pour permettre à un processus d'arrêter d'attendre des messages de processus crashés. Comme deux majorités s'intersectent toujours, on peut implémenter Σ dans un système où une majorité de processus sont corrects.

4.1.2.3 La classe de détecteurs de fautes $M\Sigma$

Définition Cette classe de détecteurs de fautes est conçue pour le modèle $SM\_MP_{n,m}[\Sigma]$. Elle comprend tous les détecteurs de fautes qui satisfont les propriété suivantes dans lesquelles le quorum $M\Sigma_i$ est la sortie locale au processus p_i et où $M\Sigma_i^\tau$ est sa valeur à l'instant τ.

- *Intersection.* $\forall\, i,j \in \Pi,\ \forall\, \tau,\tau' :\ \exists x,k,\ell :\ (x \in [1..m]) \wedge (k \in M\Sigma_i^\tau) \wedge (\ell \in M\Sigma_j^{\tau'}) \wedge (k,\ell \in P[x])$.
- *Vivacité.* $\exists\, \tau :\ \forall\, \tau' \geq \tau :\ \forall\ \in \mathcal{C} :\ M\Sigma_i^\tau \subseteq \mathcal{C}$.

La propriété de vivacité est la même que celle de Σ. La propriété d'intersection est plus générale. Elle établit que que chaque paire de quorums (dont les valeurs sont prises à des instants quelconques) est telle que chacun des deux quorum contient un processus tel que les deux processus partagent la même mémoire. Cela peut être vu comme une intersection "indirecte" : $M\Sigma_i$ et $M\Sigma_j$ ne sont pas obligés de s'intersecter directement mais ils doivent contenir des processus qui partagent la même mémoire.

Cas particuliers Considérons d'abord le cas $m = 1$ (le modèle est alors le modèle en mémoire partagée classique). Dans ce cas, il y a une seule mémoire (MEM_1) et en prenant toujours $M\Sigma_i = \{i\}$ pour chaque p_i, les deux propriétés sont toujours satisfaites. Il y a donc une implémentation triviale de $M\Sigma$ dans $SM\_MP_{n,1}[\emptyset]$, ce qui signifie que $M\Sigma$ n'ajoute pas de puissance de calcul quand $m = 1$. Cela correspond parfaitement au fait que $SM\_MP_{n,1}[\emptyset]$ est le modèle en mémoire partagée dans lequel les registres sont déjà fournis.

Considérons maintenant le cas $m = n$ (le modèle par passage de message classique). Dans ce cas, il y a un seul processus par groupe x (par exemple, $P[x]$ ne contient que p_x). Pour que la propriété d'intersection soit vraie, on doit donc avoir $\forall\, i,j \in \Pi,\ \forall\, \tau,\tau' :\ \exists k :\ (k \in M\Sigma_i^\tau) \wedge (k \in M\Sigma_j^{\tau'})$, c'est-à-dire $\forall\, i,j,\ \forall\, \tau,\tau' : M\Sigma_i^\tau \cap M\Sigma_j^{\tau'} \neq \emptyset$. Quand on considère $m = n$, $M\Sigma$ devient donc Σ, ce qui signifie que $SM\_MP_{n,n}[M\Sigma]$ et $SM\_MP_{n,n}[\Sigma]$ définissent le même modèle de calcul.

4.1.3 $M\Sigma$ est suffisant : construction d'un registre atomique dans $SM\_MP_{n,m}[M\Sigma]$

Cette section présente et démontre un algorithme qui construit un registre atomique 1WMR dans $SM\_MP_{n,m}[M\Sigma]$. Le processus écrivain est noté p_w. Le registre atomique qui est construit est noté REG.

L'algorithme, présenté dans la figure 4.2, est une adaptation simple au modèle hybride de l'algorithme présenté dans [16] qui construit un registre atomique dans un système à passage de message dans lequel une majorité de processus est correcte. Comme indiqué précédemment, l'opération send est atomique mais l'opération broadcast ne l'est pas.

Cet algorithme n'est pas conçu dans le but d'être efficace. Son but est de montrer qu'on peut construire un registre atomique dans $SM\_MP_{n,m}[M\Sigma]$, et donc que $M\Sigma$ est suffisant. (Rappelons que, quand $m = 1$, la communication par passage de message peut être simulée facilement en utilisant la mémoire partagée.)

Les variables qui implémentent le registre atomique REG Soient p_i un processus et x son groupe (c'est-à-dire que $i \in P[x]$). Le processus p_i stocke sa "copie locale" de REG dans $MEM_x[i]$. Plus

précisément, ce registre (partagé seulement entre les membres de $P[x]$) a deux champs : $MEM_x[i].val$ (qui stocke la dernière valeur de REG connue de p_i) et $MEM_x[i].sn$ (qui stocke le numéro de séquence correspondant).

L'opération $REG.\mathsf{write}(v)$ Cette opération (qui ne peut être invoquée que par p_w) commence par associer un numéro de séquence (sn_w) à son invocation actuelle (ligne 01). Il envoie ensuite le message WRITE(v, sn_w) à tous les processus pour les informer de la nouvelle écriture (ligne 02). Quand p_w a reçu un accusé de réception de la part de tous les processus de $M\Sigma_w$, l'opération renvoie ok et termine (lignes 02-04).

Soit p_i un processus tel que $i \in P[x]$. Quand p_i (p_i peut être p_w) reçoit un message WRITE$(v, seqnb)$ d'un processus p_j, il met à jour $MEM_x[i]$ si ce message contient une écriture plus récente (ligne 12). De plus, p_i envoie toujours un accusé de réception contenant $seqnb$ (ligne 13) pour informer p_j que sa "copie locale" de REG a maintenant un numéro que séquence $\geq seqnb$.

```
operation REG.write(v) :   % This code is only for the single writer p_w %
(01)  sn_w ← sn_w + 1 ;
(02)  broadcast WRITE(v, sn_w) ;
(03)  wait until (MΣ_w is such that ∀ j ∈ MΣ_w : p_w has received ACK(sn_w) from p_j) ;
(04)  return(ok).
-------------------------------------------------------------------------------
% The code snippets that follow are for every process p_i, 1 ≤ i ≤ n %
% Moreover, the value x denotes p_i's partition number i.e., x is such that i ∈ P[x] %

operation REG.read() :
(05)  r_sn_i ← r_sn_i + 1 ;
(06)  broadcast READ(r_sn_i) ;
(07)  wait until (MΣ_i is such that ∀ j ∈ MΣ_i : p_i has rec. VAL(v, sn, r_sn_i) from p_j) ;
(08)  ⟨v, sn⟩ ← (⟨v, sn⟩ | VAL(v, sn, r_sn_i) rec. ∧ ∄ sn' > sn : VAL(−, sn', −) rec.) ;
(09)  broadcast WRITE(v, sn) ;
(10)  wait until (MΣ_i is such that ∀ j ∈ MΣ_i : p_i has received ACK(sn) from p_j) ;
(11)  return(v).

Task T1 : when WRITE(v, seqnb) is received from p_j :
(12)  if (MEM_x[i].sn < seqnb) then MEM_x[i] ← ⟨v, seqnb⟩ end if ;
(13)  send ACK(seqnb) to p_j.

Task T2 : when READ(r_sn) is received from p_j :
(14)  mem ← {MEM_x[k] such that k ∈ P[x]} ;
(15)  ⟨v, sn⟩ ← (mem[k] | ∄ℓ : mem[ℓ].sn > mem[k].sn) ;
(16)  send VAL(v, sn, r_sn) to p_j.
```

FIGURE 4.2 – Construction d'un registre atomique 1WMR dans $SM\_MP_{n,m}[M\Sigma]$

L'opération $REG.\mathsf{read}()$ Cette opération est divisée en deux phases. Dans la première, (lignes 05-08), p_i broadcaste un message READ(r_sn_i) où r_sn_i est utilisé pour identifier l'invocation de l'opération de lecture (lignes 05-06) et attend que $M\Sigma_i$ contienne seulement des processus desquels p_i a reçu un accusé correspondant (ligne 07).

Quand un processus p_k reçoit un message READ(r_sn) d'un processus p_j, il calcule la valeur la plus récente de REG contenue dans la mémoire MEM_x, où $k \in P[x]$ (lignes 14-15) et renvoie à p_j cette valeur (line 16).

Finalement, p_i détermine la valeur la plus récente de REG qu'il a reçu des processus de $M\Sigma_i$ (ligne 08). Cette valeur sera renvoyée par l'opération de lecture (ligne 11), mais avant, p_i doit exécuter la

deuxième phase de l'algorithme (lignes 08-10) dont le but est de garantir qu'aucune valeur écrasée ne sera renvoyée par une opération de lecture. Pour cela, p_i simule une écriture de la valeur qu'il va renvoyer.

Théorème 4.2. *Soit $1 \leq m \leq n$. L'algorithme présenté dans la figure 4.2 est une implémentation sans-attente correcte d'un registre atomique 1WMR dans le modèle $SM\_MP_{n,m}[M\Sigma]$.*

Démonstration. Nous devons montrer que (a) chaque invocation d'une opération par un processus correct termine quel que soit le nombre de crashes (sans-attente) et que (b) toutes les invocations d'opérations (sauf éventuellement, pour chaque processus, sa dernière invocation si il est fautif) peuvent être ordonnées totalement de telle manière que (quand on considère cet ordre total) aucune opération ne renvoie une valeur écrasée et si une invocation $inv1$ termine avant qu'une opération $inv2$ ne démarre, alors $inv1$ apparait avant $inv2$ dans l'ordre total.

Démonstration de (a). Soit p_i un processus correct qui invoque une opération sur REG. On peut observer que le seul endroit où un processus pourrait être bloqué indéfiniment est durant l'instruction d'attente de messages de processus de $M\Sigma_i$. Les observations suivantes concluent la démonstration : (1) Chaque instruction d'attente suit un broadcast identifié par un numéro de séquence, (2) chaque message de broadcast reçoit un réponse de tous les processus corrects, (3) les canaux sont fiables et (4) après un temps fini, $M\Sigma_i$ ne contient que des processus corrects.

Démonstration de (b). Observons que, comme il n'y a qu'un écrivain et qu'il est séquentiel, les invocations de $REG.\mathsf{write}()$ sont ordonnées totalement et cet ordre est en accord avec leurs numéros de séquence. Initialisons donc la séquence S à la séquence de toutes les invocations d'opérations d'écriture ordonnées selon leurs numéros de séquence (cette séquence respecte leur ordre temporel).

Considérons une invocation de $REG.\mathsf{read}()$ par un processus p_i. Soit sn le numéro de séquence de la valeur renvoyée par cette invocation. Chaque invocation de $REG.\mathsf{read}()$ est ajoutée à S après l'écriture dont le numéro de séquence est sn et après (si elle existe) celle dont le numéro est $sn + 1$. De plus, si deux opérations de lecture reçoivent le même numéro, celle qui a commencé en premier est placée dans S avant l'autre. Selon la manière dont elle est définie, S est donc une histoire correcte d'exécution d'un registre (aucune lecture ne reçoit une valeur écrasée dans S).

Il reste à montrer que S respecte l'ordre temporel des invocations. Comme nous l'avons vu, c'est déjà le cas pour les opérations d'écriture. Pour les invocations de $REG.\mathsf{read}()$, il y a deux cas à considérer.

- L'invocation de l'opération de lecture commence après que la α-ième opération d'écriture ait terminé. Pour prouver qu'elle apparait dans S après cette opération de lecture, nous montrons que le numéro de séquence de cette écriture est $\geq \alpha$.
  Soient p_i le processus qui a invoqué cette opération de lecture et $M\Sigma_i^\tau$ la valeur du quorum qui lui a permis d'arrêter d'attendre à la ligne 07 de son invocation de lecture. Soit $M\Sigma_w^{\tau'}$ la valeur du quorum qui a permis à p_w d'arrêter d'attendre à la ligne 03 de son α-ième invocation d'écriture. Comme p_i a commencé à lire après que p_w ait terminé son α-ième écriture, on a $\tau > \tau'$. De plus, selon la propriété d'intersection de $M\Sigma$, $\exists x, k, \ell : \ (k \in M\Sigma_i^\tau) \wedge (\ell \in M\Sigma_w^{\tau'}) \wedge (k, \ell \in P[x])$. On a donc, d'après les lignes 12-13 exécutées par p_ℓ juste avant qu'il renvoie $\text{ACK}(\alpha)$ à p_w, $MEM_x[\ell].sn \geq \alpha$.
  Comme $\tau > \tau'$ et $k \in M\Sigma_i^\tau$, on a aussi que quand p_k exécute les lignes 14-15 (exécution provoquée par la réception du message $\text{READ}(r\_sn)$ de p_i), $MEM_x[\ell].sn \geq \alpha$. p_k envoie donc à p_i le message $\text{VAL}(-, \beta, r\_sn)$ où $\beta \geq \alpha$ (ligne 16). Le numéro de séquence calculé par p_i à la ligne 07 de l'opération de lecture est donc $\geq \alpha$ ce qui prouve ce cas.
- Le deuxième cas est quand deux opérations de lecture non concurrentes sont telles que la première reçoit le numéro de séquence $sn1$ et la deuxième reçoit $sn2$ (ces invocations séquentielles peuvent venir de processus différents et être en concurrence avec des opérations de lecture). Nous devons montrer que $sn1 \geq sn2$.
  Soit p_j le processus qui a invoqué la première opération de lecture. Le raisonnement est le même que celui du cas précédent après avoir remplacé l'écriture de p_w par les lignes 09-10 exécutées par p_j durant son opération de lecture.

□

4.1.4 $M\Sigma$ est nécessaire

4.1.4.1 $M\Sigma$ est le plus faible détecteur de fautes pour un registre dans un modèle de communication hybride

La section précédente a montré qu'un registre atomique peut être construit dans $SM\_MP_{n,m}[M\Sigma]$, montrant ainsi qu'enrichir $SM\_MP_{n,m}[\emptyset]$ avec $M\Sigma$ est suffisant (d'un point de vue "information sur les fautes") quand on veut construire un registre atomique. Cette section considère la nécessité. Elle montre que tout détecteur de fautes D tel qu'un registre atomique peut être construit dans $SM\_MP_{n,m}[D]$ fournit suffisamment d'information sur les fautes pour implémenter $M\Sigma$ dans $SM\_MP_{n,m}[D]$.

Soit D un détecteur de fautes quelconque tel qu'il existe un algorithme A qui construit un registre atomique dans $SM\_MP_{n,m}[D]$. La démonstration de la partie "nécessité" consiste à montrer qu'il est possible de construire un détecteur de fautes de la classe $M\Sigma$ à partir de A exécuté dans $SM\_MP_{n,m}[D]$. Dans le jargon des détecteurs de fautes, nous disons qu'il est possible d'"extraire" Σ de A. D'une manière très intéressante, l'algorithme d'extraction proposé est le même que celui présenté dans [26, 110] (pour Σ) mais sa démonstration est différente. Nous montrons donc ici que l'algorithme d'extraction présenté dans [26] a une dimension générique par rapport aux détecteurs de fautes.

4.1.4.2 L'algorithme d'extraction de Bonnet-Raynal

Cette section présente l'algorithme d'extraction de Bonnet-Raynal introduit dans [26] où il est considéré que l'algorithme A basé sur D construit un registre atomique dans $SM\_MP_{n,m}[D]$.

Tableaux de registres atomiques Soient Q un ensemble non vide de processus et $REG_Q[1..n]$ un tableau de n registres atomiques (initialisé à $[\bot, \ldots, \bot]$) tel que chaque registre atomique $REG_Q[x]$ est implémenté par l'algorithme pour n processus A exécuté uniquement par $|Q|$ fils d'exécution, chacun associé à un processus de Q.

Un algorithme simple basé sur des registres Soit WR_Q l'algorithme (aussi appelé tâche) dans lequel chaque processus p_i tel que $i \in Q$ exécute le code suivant (où $reg_i[1..n]$ est un tableau local à p_i) :

algorithm WR_Q :
$REG_Q[i]$.write($\top$) ; **for each** $x \in \{1, ..., n\}$ **do** $reg_i[x] \leftarrow REG_Q[x]$.read() **end for**.

Le processus p_i commence par écrire la valeur $\top$ dans son entrée du tableau REG_Q, puis lit de manière asynchrone toutes les entrées. Les opérations $REG_Q[i]$.write($\top$) et $REG_Q[x]$.read() sont fournies aux processus par l'algorithme A. (Observons que la valeur lue n'a pas d'importance. Comme nous le verrons, ce qui importe est le fait que $REG_Q[x]$ ait été écrit ou non.) Une exécution de WR_Q est notée E_Q. Dans cette exécution, aucun processus n'appartenant pas à Q n'envoie ni ne reçoit de messages liés à l'algorithme WR_Q.

Rappelons que $\mathcal{C}$ désigne l'ensemble des processus corrects dans l'exécution considérée. Observons que, comme l'algorithme A basé sur le détecteur de fautes D est correct, si l'ensemble Q contient tous les processus corrects (c'est-à-dire $\mathcal{C} \subseteq Q$), tous les processus corrects de E_Q terminent la tâche WR_Q. Dans le cas contraire, c'est-à-dire pour les tâches WR_Q telles que $\neg(\mathcal{C} \subseteq Q)$, E_Q est telle qu'un processus de Q soit termine WR_Q, soit est bloqué indéfiniment, soit crashe. (Cela dépend du motif de fautes, des sorties du détecteur de fautes D utilisé par A et du code de A. Par exemple, considérons la tâche WR_Q et deux processus corrects p_i et p_j tels que $i \in Q$ et $j \notin Q$. Soit $th_{i,Q}$ le fil d'exécution de p_i dans Q. Comme $j \notin Q$, le fil $th_{j,Q}$ n'existe pas. Le fil $th_{i,Q}$ peut alors être bloqué indéfiniment quand il exécute

A pour lire ou écrire un registre de $REG_Q[1..n]$ si, à cause de la sortie de D et du code de A, il doit attendre un message de $th_{j,Q}$, qui n'existe pas [2]).

Exécution de $2^n - 1$ tâches concurrentes L'algorithme d'extraction considère les $2^n - 1$ tâches distinctes WR_Q où Q est un ensemble non vide de 2^Π. Pour cela, chaque processus p_i gère 2^{n-1} fils d'exécution, un pour chaque sous-ensemble Q tel que $i \in Q$. Remarquons que le crash d'un processus p_i entraine le crash de tous ses fils d'exécution.

L'algorithme d'extraction L'algorithme qui extrait Σ est présenté dans la figure 4.3. Rappelons que le but est de fournir à chaque processus p_i une variable locale telle que les variables $(\Sigma_x)_{1 \leq x \leq n}$ satisfont les propriétés d'intersection et de vivacité de Σ.

Pour cela, chaque processus p_i gère deux variables : un ensemble d'ensembles d'identités de processus, noté $quorum\_sets_i$, et une file notée $queue_i$. Le but de l'ensemble d'ensembles $quorum\_sets_i$ est de contenir tous les ensembles Q tels que p_i termine WR_Q (tâche $T1$), tandis que $queue_i$ est gérée de manière à ce que tous les processus corrects finissent par apparaitre avant les processus fautifs (tâches $T2$ et $T3$).

L'idée est de sélectionner un élément de $quorum\_sets_i$ comme sortie pour Σ_i. Comme nous le verrons dans la démonstration, étant donnée une paire quelconque de processus p_i et p_j, chaque quorum de $quorum\_sets_i$ a une intersection non vide avec chaque quorum de $quorum\_sets_j$, fournissant donc la propriété d'intersection requise.

La difficulté principale est de garantir la propriété de vivacité de Σ_i (après un temps fini, Σ_i ne doit contenir que des processus corrects) tout en préservant la propriété d'intersection. On arrive à ce but avec l'aide de la variable locale $queue_i$ de la manière suivante : la sortie de Σ est l'ensemble (quorum) de $quorum\_sets_i$ qui apparait comme étant le 'premier" dans $queue_i$. La définition formelle de 'premier élément de $quorum\_sets_i$ par rapport à $queue_i$" est établie dans la tâche $T4$. Pour faciliter sa compréhension, considérons l'exemple suivant. Soient $quorum\_sets_i = \{\{3,4,9\},\{2,3,8\},\{1,2,4,7\}\}$ et $queue_i = < 4,8,3,2,7,5,9,1,\cdots >$. L'ensemble $S = \{2,3,8\}$ est le premier ensemble de $quorum\_sets_i$ par rapport à $queue_i$ parce que chacun des autres ensembles $\{3,4,9\}$ et $\{1,2,4,7\}$ inclut un élément (respectivement 9 et 7) qui apparait dans $queue_i$ après les éléments de S. (Au cas où plusieurs ensembles sont 'premiers", on peut prendre n'importe lequel d'entre eux). La notion de *premier quorum* est utilisée pour garantir la vivacité de Σ, c'est-à-dire qu'après un temps fini, l'ensemble Σ_i de chaque processus correct p_i ne contient que des processus corrects.

Remarque 1 Initialement, $quorum\_sets_i$ contient l'ensemble $\{1,\ldots,n\}$. Comme aucun ensemble n'est jamais retiré de $quorum\_sets_i$ (tâche $T1$), $quorum\_sets_i$ n'est jamais vide. De plus, il n'est pas nécessaire de lancer $WR_{\{1,\ldots,n\}}$ Dans laquelle tous les processus participent. Cela est dû au fait que, comme l'algorithme A (qui implémente un registre) est correct, tous les processus terminent dans la tâche $WR_{\{1,\ldots,n\}}$. Ce cas est directement pris en compte à l'initialisation de $quorum\_sets_i$ (évitant ainsi l'exécution de la tâche $WR_{\{1,\ldots,n\}}$).

Remarque 2 En observant l'algorithme d'extraction, on peut remarquer que (1) les deux variables $queue_i$ et $quorum\_sets_i$ sont bornées et que (2) les messages contiennent des valeurs bornées ; la construction est donc bornée.

2. Un blocage similaire peut arriver quand les processus utilisent un algorithme basé sur Ω [38] (le plus faible détecteur de fautes permettant de résoudre le problème du consensus en mémoire partagée) et que, dans l'exécution considérée, le processus correct qui est finalement élu ne participe pas à l'algorithme.

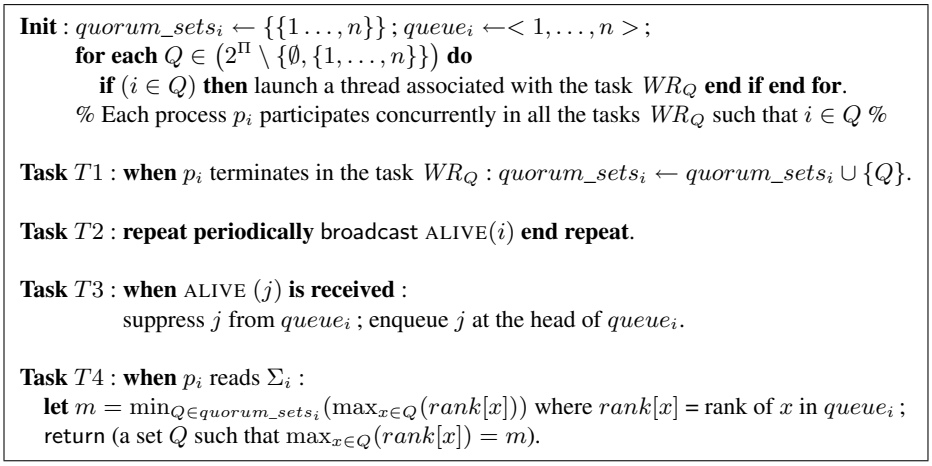

FIGURE 4.3 – Extraction de Σ à partir d'un algorithme A basé sur un détecteur de fautes qui implémente un registre (code de p_i)

4.1.4.3 Minimalité de $M\Sigma$

Théorème 4.3. *Soit $1 \leq m \leq n$. $M\Sigma$ est le plus faible détecteur de fautes avec lequel $SM\_MP_{n,m}[\emptyset]$ doit être enrichi pour pouvoir construire un registre atomique.*

Démonstration. Soit D un détecteur de fautes quelconque tel qu'il existe un algorithme A qui construit un registre atomique dans $SM\_MP_{n,m}[D]$. La preuve consiste à montrer que, étant donné un tel algorithme A, l'algorithme présenté dans la figure 4.3 construit un détecteur de fautes $M\Sigma$.

Démonstration de la propriété d'intersection
La démonstration est par contradiction. Observons d'abord que que l'ensemble Σ_i renvoyé à un processus p_i est un ensemble de $quorum\_set_i$ (qui contient l'ensemble $\{1,\ldots,n\}$ -valeur initiale- et tous les ensembles Q tels que p_i termine WR_Q). Supposons qu'il y a deux ensembles Q_1 et Q_2 tels que (1) $Q_1, Q_2 \in \bigcup_{1 \leq j \leq n}(quorum\_set_j)$ et (2) $\forall\, x, k, \ell : (k \in Q_1 \wedge \ell \in Q_2) \Rightarrow (\{k, \ell\} \not\subseteq P[x])$. Remarquons que (1) signifie que Q_1 et Q_2 peuvent être renvoyés à des processus comme valeur locale de Σ. (2) signifie qu'au moins un de k et ℓ n'est pas dans $P[x]$, d'où on en conclue que les processus de Q_1 et Q_2 ne peuvent pas communiquer par la mémoire partagée $P[x]$.

Soit p_i un processus qui termine WR_{Q_1} et p_j un processus qui termine WR_{Q_2} (à cause de l'hypothèse qui mènera à la contradiction, ces processus existent). En utilisant le fait que le système est asynchrone, on construit les exécutions E_{Q_1} et E_{Q_2} associées à WR_{Q_1} et WR_{Q_2} de la manière suivante. S'ils existent, les messages envoyés par les processus de Q_1 aux processus de Q_2, quand ils exécutent l'algorithme A pour implémenter chaque registre du tableau REG_{Q_1}, sont retardés pour une période arbitrairement longue (jusqu'à ce que p_i ait ajouté Q_1 à $quorum\_set_i$ et p_j ait ajouté Q_2 à $quorum\_set_j$). Idem pour les messages envoyés par les processus de Q_1 aux processus de Q_2 quand ils exécutent l'algorithme A pour implémenter chaque registre du tableau REG_{Q_1}.

Observons que, dans les exécutions concurrentes E_{Q_1} et E_{Q_2}, l'algorithme A, qui est exécuté (1) uniquement par les processus de Q_1 dans E_{Q_1} pour construire les registres $REG_{Q_1}[1..n]$ et (2) uniquement par les processus de Q_2 dans E_{Q_2} pour construire les registres $REG_{Q_2}[1..n]$, reçoit les mêmes sorties du détecteur de fautes D. Comme (a) $p_i \in Q_1$ et $p_j \in Q_2$, et (b) $\forall\, x, k, \ell : (k \in Q_1 \wedge \ell \in Q_2) \Rightarrow (\{k, \ell\} \not\subseteq P[x])$, p_i n'écrit pas dans $REG_{Q_2}[i]$ et p_j n'écrit pas dans $REG_{Q_1}[j]$. p_i lit donc $\perp$ dans $REG_{Q_1}[j]$ et p_j lit $\perp$ dans $REG_{Q_2}[i]$.

Construisons une exécution $E_{Q_{12}}$ dans laquelle $Q_{12} = Q_1 \cup Q_2$, c'est-à-dire une combinaison de E_{Q_1} et E_{Q_2} définie de la manière suivante. Dans cette exécution, l'algorithme A (dans lequel seuls les processus de Q_{12} participent et qui implémente le tableau de registres $REG_{Q_{12}}[1..n]$) reçoit les

mêmes sorties du détecteur de fautes D que celles fournies aux exécutions concurrentes E_{Q_1} et E_{Q_2}. De plus, les messages de Q_1 vers Q_2 et de Q_2 vers Q_1 sont retardés comme dans E_{Q_1} et E_{Q_2}. p_i (respectivement p_j) reçoit les mêmes messages et les mêmes sorties du détecteur de fautes D dans $E_{Q_{12}}$ et E_{Q_1} (respectivement E_{Q_2}).

- D'un coté, le processus p_i reçoit les mêmes messages et les mêmes sorties du détecteur de fautes D dans $E_{Q_{12}}$ et E_{Q_1}. $REG_{Q_1}[1..n]$ et $REG_{Q_{12}}[1..n]$ contiennent donc les mêmes valeurs. p_i lit donc $\perp$ dans $REG_{Q_{12}}[j]$. De la même manière, p_j lit $\perp$ dans $REG_{Q_{12}}[i]$.
- D'un autre coté, dans $E_{Q_{12}}$, le processus p_i écrit $\top$ dans $REG_{Q_{12}}[i]$ et le processus p_j écrit $\top$ dans $REG_{Q_{12}}[j]$. De plus, l'une de ces opérations termine avant l'autre. Sans perte de généralité, considérons que l'écriture de p_i termine avant l'écriture de p_j. p_j lit donc $REG_{Q_{12}}[i]$ après qu'il a écrit. À cause de la propriété d'atomicité du registre, p_j obtient donc la valeur $\top$ quand il lit $REG_{Q_{12}}[i]$.

Le deuxième point contredit le premier. La supposition initiale (existence d'un algorithme A basé sur un détecteur de fautes qui construit un registre, $Q_1, Q_2 \in \bigcup_{1\leq j\leq n}(quorum\_set_j)$ et $\forall\, x, k, \ell : (k \in Q_1 \wedge \ell \in Q_2) \Rightarrow (\{k, \ell\} \not\subseteq P[x])$) est donc fausse, d'où on en conclut qu'au moins une des deux hypothèses établies au début de la démonstration (c'est-à-dire (1) $Q_1, Q_2 \in \bigcup_{1\leq j\leq n}(quorum\_set_j)$ et (2) $\forall\, x, k, \ell : (k \in Q_1 \wedge \ell \in Q_2) \Rightarrow (\{k, \ell\} \not\subseteq P[x])$) est fausse, ce qui conclut la démonstration de la propriété d'intersection de $M\Sigma$.

Démonstration de la propriété de vivacité
Pour la propriété de vivacité, considérons la tâche $WR_{\mathcal{C}}$ (rappelons que $\mathcal{C}$ est l'ensemble des processus corrects). Comme l'algorithme A basé sur un détecteur de fautes qui implémente les registres $REG_{\mathcal{C}}[1..n]$ est correct, chaque processus correct p_i termine ses opérations $REG_{\mathcal{C}}[i].\mathsf{write}(\top)$ et $REG_{\mathcal{C}}[x].\mathsf{read}()$ dans $E_{\mathcal{C}}$. Dans l'algorithme d'extraction, après un temps fini, la variable $quorum\_set_i$ de chaque processus correct p_i contient donc l'ensemble $\mathcal{C}$.

De plus, après un temps fini, chaque processus p_i ne reçoit des messages $\text{ALIVE}(j)$ que de la part de processus corrects. Cela signifie que, pour chaque processus correct p_i, après un temps fini, tous les processus corrects précéderont les processus fautifs dans $queue_i$. À cause de la définition de "premier ensemble de $quorum\_set_i$ dans $queue_i$" établie dans la tâche $T4$, à partir du moment où $\mathcal{C}$ a été ajouté à $quorum\_set_i$, le quorum Q sélectionné par la tâche $T4$ est donc toujours tel que $Q \subseteq \mathcal{C}$, ce qui conclut la démonstration de la propriété de vivacité de $M\Sigma$. □

4.1.4.4 $M\Sigma$ est strictement plus faible que Σ quand $m < n$

Théorème 4.4. *Soit $m < n$. Le modèle $SM\_MP_{n,m}[M\Sigma]$ est strictement plus faible que le modèle $SM\_MP_{n,m}[\Sigma]$.*

Démonstration. Pour démontrer le théorème, nous devons montrer que (a) on peut construire $M\Sigma$ dans $SM\_MP_{n,m}[\Sigma]$ et (b) il est impossible de construire Σ dans $SM\_MP_{n,m}[M\Sigma]$ quand $m < n$.

- Démonstration de (a). Pour chaque i et τ, définissons $M\Sigma_i^\tau = \Sigma_i^\tau$. Comme $\Sigma_i^\tau \cap \Sigma_j^{\tau'} \neq \emptyset$, on a $\exists k \in M\Sigma_i^\tau \cap M\Sigma_j^{\tau'}$ et il y a une partition x telle que $k \in P[x]$ ce qui prouve la propriété d'intersection de $M\Sigma$. La propriété de vivacité de $M\Sigma$ est obtenue directement avec celle de Σ.
- Démonstration de (b). La démonstration est par contradiction. Supposons qu'il existe un algorithme sans-attente A qui construit Σ dans $SM\_MP_{n,m}[M\Sigma]$ quand $m < n$.
  Comme $m < n$, il y a une partition $P[x]$ et une paire de processus p_i et p_j telles que $i, j \in P[x]$ (c'est-à-dire que p_i et p_j appartiennent au même groupe x). Considérons une exécution dans laquelle p_i et p_j sont corrects et tous les autres processus crashent avant leur première instruction. Comme $i, j \in P[x], \forall\, \tau, \tau'$, $M\Sigma_i^\tau = \{i\}$ et $M\Sigma_j^{\tau'} = \{j\}$ sont des sorties correctes de $M\Sigma$ (elles satisfont ses propriétés d'intersection et de vivacité).

Supposons que, pendant son exécution de A, p_j fait une pause pendant un temps fini mais arbitrairement long durant lequel p_i s'exécute seul et (à cause de l'asynchronie) ne reçoit pas de message de p_j. Comme $\forall\ \tau$ on a $M\Sigma_i^\tau = \{i\}$, p_i ne peut pas distinguer cette exécution de A d'une exécution dans laquelle il est le seul processus correct. Après un temps fini, parce qu'il est sans-attente, A devra donc fournir comme sortie $\{i\}$ à p_i pour satisfaire la propriété de vivacité de Σ. Il y a donc un instant τ_i tel que $\Sigma_i^{\tau_i} = \{i\}$.
Supposons maintenant que, après l'instant τ_i, p_i fait une pause pendant un temps fini mais arbitrairement long durant lequel p_j s'exécute seul et (à cause de l'asynchronie) ne reçoit pas de message de p_i. Selon le même raisonnement que précédemment, il y a un instant τ_j auquel on a $\Sigma_j^{\tau_j} = \{j\}$. On a donc $\Sigma_i^{\tau_i} \cap \Sigma_j^{\tau_j} = \emptyset$ et la propriété d'intersection de Σ est violée, ce qui conclut la démonstration de théorème.

□

Remarque Observons que, d'après la deuxième partie (b) du théorème précédent, quand les processus de tous les groupes sauf un crashent, $M\Sigma$ est trop faible pour donner des informations sur les fautes. Le corollaire suivant est une implication du théorème précédent quand on considère le cas $m = 1$ (modèle à mémoire partagée classique dans lequel $M\Sigma$ peut être implémenté de façon triviale).

Corollaire 4.5. *Σ ne peut pas être construit uniquement à partir de registres.*

4.1.5 Sur la possibilité d'implémenter $M\Sigma$ malgré l'asynchronie et les fautes

Quand $m = n$ (système à passage de message classique), $M\Sigma$ est en fait Σ et on sait que Σ peut être implémenté dans un système à passage de message classique quand une majorité de processus sont corrects. D'où la question suivante : existe-t-il une condition nécessaire et suffisante C sur n, m et un paramètre du système associé aux fautes tels que $M\Sigma$ peut être implémenté dans $SM\_MP_{n,m}[C]$ (où $SM\_MP_{n,m}[C]$ désigne le modèle $SM\_MP_{n,m}[\emptyset]$ restreint aux exécutions où C est satisfaite) ? Cette section présente une telle condition nécessaire et suffisante C.

Notion de groupe fautif Nous disons qu'un groupe x est fautif dans une exécution si tous les processus de $P[x]$ sont fautifs dans cette exécution. Soit t, $1 \leq t < m$ la borne supérieure sur le nombre de groupes fautifs.

Le théorème suivant montre que $C = (t < m/2)$ est une condition nécessaire et suffisante pour implémenter $M\Sigma$ dans $SM\_MP_{n,m}[\emptyset]$.

Théorème 4.6. *Soient $COND$ l'ensemble de tous les prédicats n, m et t, $C' \in COND$ et $C = (t < m/2)$. $M\Sigma$ peut être construit dans $SM\_MP_{n,m}[C']$ si et seulement si $C' \Rightarrow C$.*

Démonstration. La démonstration est en deux parties : (a) $M\Sigma$ peut être construit dans toutes les exécutions dans lesquelles C est satisfaite et (b) $M\Sigma$ ne peut pas être construit dans toutes les exécutions dans lesquelles C n'est pas satisfaite.

Démonstration de (a). L'algorithme décrit dans la figure 4.4 construit $M\Sigma$ dans $SM\_MP_{n,m}[t < m/2]$. À l'initialisation, chaque processus p_i initialise $M\Sigma_i$ à Π (l'ensemble de toutes les identités de processus). Ensuite, de manière répétitive, p_i broadcaste un message ALIVE(i), attend jusqu'à ce qu'il ait reçu des messages de $(m - t)$ processus appartenant à des groupes différents et affecte cet ensemble de processus à $M\Sigma_i$. On montre maintenant que les propriétés d'intersection et de vivacité de $M\Sigma$ sont satisfaites.

– Observons d'abord que, à cause de l'hypothèse $t < m/2$, aucun processus ne peut rester bloqué dans l'instruction d'attente. De plus, après un temps fini, un processus correct ne reçoit des

messages que de la part d'autres processus corrects. Ces deux observations impliquent que, après un temps fini, $M\Sigma_i$ ne contient que des processus corrects, ce qui constitue la propriété de vivacité de $M\Sigma$.

– $\forall\ i, j\ \in\ \Pi,\ \forall\ \tau, \tau'$, considérons les valeurs de $M\Sigma_i^\tau$ et de $\Sigma_j^{\tau'}$. $t\ <\ m/2$ implique que Σ_i^τ contient des processus appartenant à une majorité de groupes ; idem pour $\Sigma_j^{\tau'}$. Comme deux majorités s'intersectent toujours, on peut conclure qu'il existe un groupe x tel que $k \in M\Sigma_i^\tau \wedge \ell \in M\Sigma_j^{\tau'} \wedge \{k, \ell\} \subseteq P[x]$, ce qui constitue la propriété d'intersection de $M\Sigma$.

```
M Σ_i ← Π ;
repeat forever
        broadcast ALIVE(i) ;
        wait until (messages received from processes in (m − t) different clusters) ;
        M Σ_i ← the set of processes from which messages have been received at previous line
end repeat.
```

FIGURE 4.4 – Construction de $M\Sigma$ dans $SM\_MP_{n,m}[t < m/2]$ (code de p_i)

Démonstration de (b). Considérons que $t \geq m/2$ et partitionnons l'ensemble de groupes en deux sous-ensembles QC_1 et QC_2 (c'est-à-dire que $QC_1 \cap QC_2 = \emptyset$ et $QC_1 \cup QC_2 = \cup_{1 \leq x \leq m} P[x]$). À cause de l'asynchronie, on peut retarder durant un temps fini mais arbitrairement long tous les messages des processus de QC_1 vers les processus de QC_2 et tous les messages des processus de QC_2 vers les processus de QC_1. Les processus de QC_1 ne pourront alors pas distinguer le cas où les processus de QC_2 ont crashé du cas où ils sont juste très lents. Idem pour les processus de QC_2 par rapport aux processus de QC_1. L'impossibilité vient de cet argument classique de partitionnement. □

4.1.6 Conclusion

Cette section a d'abord introduit un nouveau modèle avec une communication hybride. D'un coté, chaque paire de processus peut communiquer par passage de message asynchrone. D'un autre coté, les processus sont partitionnés en groupes et, dans chaque groupe, les processus peuvent communiquer à travers une mémoire partagée.

Elle a ensuite étudié l'information sur les fautes minimale qui permet d'implémenter un registre atomique dans un tel modèle à communication hybride. Cette information minimale peut être capturée par un nouveau détecteur de fautes noté $M\Sigma$ (qui généralise le détecteur de fautes Σ). Finalement, elle a présenté une condition nécessaire et suffisante pour implémenter $M\Sigma$ malgré les effets de l'asynchronie et des fautes.

4.2 Snapshot partiel

La base de la synchronisation en mémoire partagée est la manipulation de registres, c'est-à-dire leur lecture et leur modification. Quand on considère des registres atomiques, ces opérations, prises individuellement, apparaissent comme ayant lieu à un instant précis ; il y a un ordre total entre elles et on n'a pas à se soucier de l'entrelacement de deux opérations sur un registre.

La lecture de plusieurs registres, en revanche, n'est pas atomique ; un processus doit lire ces registres individuellement. Rien ne garantit donc qu'un registre lu ne sera pas modifié avant la lecture des autres. Pour certains algorithmes, il peut être intéressant de pouvoir lire plusieurs registres de manière atomique, c'est-à-dire sans entrelacement avec les opérations d'autres processus. C'est le but d'un objet snapshot. Un tel objet consiste en un ensemble de registres et offre aux processus deux opérations : update() et snapshot(). L'opération update(v, r) affecte, de manière atomique, la valeur v au registre r. L'opération snapshot() renvoie, également de manière atomique, les valeurs de l'ensemble des registres de l'objet.

Dans certains cas, la lecture atomique de l'ensemble des registres n'est pas nécessaire ; la lecture d'un sous-ensemble de registres peut suffire. On peut évidemment filtrer la sortie de l'opération snapshot() pour obtenir les valeurs voulues, mais il est alors intéressant de rechercher une solution plus efficace. C'est dans ce but qu'a été créé l'objet snapshot partiel, introduit dans [22]. Comme l'objet snapshot, il offre aux processus deux opérations : update() et p_snapshot(). L'opération update() est la même que pour l'objet snapshot classique. L'opération p_snapshot() prend en paramètre un ensemble R de registres et renvoie, de manière atomique, l'ensemble des valeurs de ces registres.

Cette section présente des implémentations efficaces d'objets snapshot partiels multi-lecteurs/multi-écrivains dans le modèle asynchrone fournissant des registres atomiques et des objets *ensemble actif* [8]. Les objets ensemble actif offrent trois opérations : Join(), Leave() et GetSet(). Intuitivement, Join() ajoute le processus qui l'invoque à l'ensemble actif et Leave() l'enlève de cet ensemble. GetSet() renvoie la valeur de l'ensemble. (Il existe des implémentations de l'objet ensemble actif n'utilisant que des registres atomiques qui sont adaptatives, c'est-à-dire dont le nombre d'accès à la mémoire ne dépend que du nombre de processus qui invoquent Join() et Leave() [8]. Le modèle utilisé dans cette section est donc le modèle asynchrone classique où les processus n'utilisent pour communiquer que des registres atomiques.)

Caractéristiques des algorithmes proposés Les algorithmes présentés ici ont plusieurs caractéristiques notables qui les distinguent des algorithmes de snapshot et de snapshot partiel proposés jusqu'ici. Ces caractéristiques sont la stratégie "écrire d'abord, aider ensuite" et la séparation entre l'écriture de la valeur et l'aide. Elles ont comme conséquences de permettre aux algorithmes de respecter les propriétés de *fraicheur* et de *localité de l'aide*.

Fraicheur Le but de la propriété de *fraicheur* est de forcer une opération update qui aide une opération snapshot à fournir à cette opération des valeurs aussi récentes que possible. Plus précisément, soient $\mathsf{up} = \mathsf{update}(r, v, i)$ une opération update invoquée par le processus p_i pour écrire la valeur v dans l'entrée r de l'objet snapshot partiel et $\mathsf{psp} = \mathsf{p\_snapshot}(R)$ une opération concurrente telle que psp démarre après up (où 'commence" signifie 'accède à la mémoire partagée pour la première fois"). La fraicheur requiert que la valeur renvoyée pour l'entrée r soit v ou une valeur plus récente (comme chaque composant est un registre atomique, la notion de 'plus récente" est bien définie) [3]. Exprimée différemment, la propriété de fraicheur force une opération update à être linéarisée au moment de son premier accès à la mémoire partagée.

Pour satisfaire cette propriété, l'algorithme de l'opération update proposé ici utilise la stratégie "écrire d'abord, aider ensuite" (les autres algorithmes proposés jusqu'ici utilisent la stratégie "aider d'abord, écrire ensuite").

Localité de l'aide Cette propriété vise à obtenir des opérations update plus efficaces en limitant la contention. Pour cela, elle réduit l'aide que peut fournir une opération update à une opération snapshot. Cette propriété est proche de la propriété "disjoint-access parallelism" définie par Israeli et Rappoport [96]. Elle diffère du "disjoint-access parallelism" en ne plaçant pas de limite sur la complexité des opérations et en étant plus restrictive sur la contention qu'elle admet. Le but de la localité de l'aide est que les accès aux objets de base par deux opérations $\mathsf{update}(r, v, -)$ et $\mathsf{p\_snapshot}(R)$ ne soient pas en conflit si $r \notin R$.

Comme pour la fraicheur, la localité de l'aide n'est pas assurée par les algorithmes proposés jusqu'ici. L'algorithme présenté dans [1] est très conservateur : chaque opération update doit calculer un snapshot entier même quand il n'y a pas d'opération snapshot concurrente. L'algorithme de snapshot partiel

3. À notre connaissance, aucun des algorithmes de snapshot proposés jusqu'ici ne satisfait la propriété de fraicheur. Ils fournissent tous au processus qui invoque le snapshot une valeur strictement plus vieille que v.

présenté dans [22] est un peu moins conservateur : une opération update$(r, -, -)$ qui n'est concurrente avec aucune opération snapshot n'est pas obligée d'aider, mais une opération update$(r, -, -)$ qui est concurrente avec des opérations p_snapshot(R_ℓ) ($1 \leq \ell \leq z$) doit aider chacune d'elles quels que soient les ensembles d'entrées auxquels elles accèdent, même les opérations p_snapshot(R_ℓ) qui n'accèdent pas à l'entrée écrite par l'opération update$(r, -, -)$.

Une caractéristique supplémentaire liée à l'asynchronie Une caractéristique supplémentaire de l'algorithme proposé concerne l'asynchronie et la taille des valeurs d'aide qu'il calcule. Les implémentations de snapshot (complet ou partiel) précédentes utilisent un registre de base $REG[r]$ par entrée r. Ces registres doivent être grands. Ils contiennent plusieurs champs, dont un pour la dernière valeur écrite, un pour une valeur de snapshot utilisée pour aider des opérations snapshot et d'autres pour des données de contrôle. Une valeur de snapshot est constituée d'une valeur pour chaque entrée de l'objet dans le cas d'un snapshot complet ou d'une valeur par entrée d'un sous-ensemble de toutes les entrées dans le cas d'un snapshot partiel. Chaque opération update$(r, v, -)$ écrit alors dans $REG[r]$, de manière atomique, la nouvelle valeur r et une valeur de snapshot. Cela implique que l'implémentation de cette écriture atomique peut être couteuse en temps et en espace.

Par contre, grâce à la stratégie "écrire d'abord, aider ensuite (individuellement)", l'algorithme de l'opération update$(r, v, -)$ présenté ici sépare l'écriture d'une valeur v dans $REG[r]$ des écritures individuelles des valeurs de snapshot d'aide, une pour chaque opération p_snapshot(R_ℓ) concurrente telle que $r \in R_\ell$. Le fait qu'une opération update$(r, v, -)$ écrive d'abord v puis aide, plus tard et individuellement, chaque opération snapshot partiel concurrente qui lit l'entrée r (1) leur permet d'obtenir une valeur de r qui est au moins aussi récente que v et (2) permet l'utilisation de plusieurs registres atomiques d'aide qui sont écrits individuellement (permettant ainsi des écritures atomiques plus efficaces). De plus, la taille de ces registres atomiques d'aide peut être plus petite que m [4].

Registres atomiques et registres LL/SC Dans cette section, on se base principalement sur un système dans lequel les processus communiquent à travers des registres atomiques mais on considère aussi le cas où les processus ont accès à des registres LL/SC. Étant donné un registre LL/SC X, l'opération $X.\mathsf{LL}()$ renvoie la valeur de X et l'opération $X.\mathsf{SC}(v)$ est une écriture qui peut soit réussir, soit échouer. Soit p le processus qui invoque $X.\mathsf{SC}(v)$. Cette écriture réussit si aucun autre processus n'a réussi à écrire dans X depuis la dernière opération $X.\mathsf{LL}()$ invoquée par p.

Ces objets de communication (qui, du point de vue de la calculabilité, sont plus puissants que des registres atomiques) permettent d'implémenter un algorithme de snapshot partiel qui est plus efficace que l'algorithme utilisant uniquement des registres atomiques, quand l'efficacité est mesurée par le nombre de registres nécessités par l'algorithme. L'algorithme basé sur des registres LL/SC a besoin de $O(n)$ registres LL/SC tandis que l'algorithme basé sur des registres atomiques classiques a besoin de $O(n^2)$ registres atomiques.

4.2.1 Modèle de système utilisé

Processus asynchrones et modèle de crash Le système est un système classique de processus asynchrones communiquant par mémoire partagé comme défini dans le chapitre 2. Il est constitué de n processus asynchrones notés $p_1, \ldots, p_n$ Un processus exécute une séquence d'étapes définie par son algorithme. Il exécute son algorithme correctement jusqu'au moment où il crashe (s'il crashe). Après avoir crashé, un processus n'exécute plus aucune étape. Dans une exécution, un processus qui crashe est dit *fautif*. Sinon, il est dit *correct*.

4. Si chaque snapshot partiel pris par un processus p_i concerne au plus k entrées, les registres atomiques d'aide utilisés pour aider p_i ont seulement besoin de contenir k entrées. Si $k << m$, les écritures dans ces registres de taille k peuvent générer moins de contention que des écritures dans des registres atomiques de taille m.

Objet ensemble actif On considère que les processus ont accès à des objets *ensemble actif*. Un tel objet, introduit dans [8], peut être utilisé pour résoudre des problèmes de synchronisation adaptatifs (voir par exemple [115]). Comme indiqué précédemment, son but est de permettre aux processus de connaitre ceux d'entre eux qui sont en train d'exécuter des opérations concurrentes. Pour cela, un objet ensemble actif offre aux processus trois opérations : Join(), Leave() et GetSet() (décrites de manière informelle au début de la section). Ces opérations ne sont pas obligatoirement atomiques.

Soit S un objet ensemble actif. Initialement, le prédicat $i \notin S$ est vrai pour chaque processus p_i (S est vide). Un processus p_i exécute la séquence d'invocations d'opérations suivante (exprimée sous la forme d'une expression régulière) : $\big[\mathsf{GetSet}()^*\big(\mathsf{Join}()\mathsf{Leave}()\big)^*\big]^*$.

- Considérons deux opérations consécutives Join() et Leave() (s'il y en a une) invoquées par un processus p_i. (Cela signifie que (1) Leave() suit immédiatement Join(), ou (2) Join() suit Leave() et il ne peut y avoir que GetSet()* entre elles.)
  - Du dernier évènement de Join() jusqu'au premier évènement de Leave(), le prédicat $i \in S$ est vrai.
  - Du dernier évènement de Leave() jusqu'au premier évènement de Join(), le prédicat $i \in S$ est faux.
  - Durant l'exécution de Join() ou Leave(), le prédicat peut être vrai ou faux.
- Une invocation de GetSet() renvoie un ensemble d'identités de processus qui inclut :
  - Chaque j tel que le prédicat $j \in S$ était vrai durant toute l'exécution de GetSet().
  - Aucun j tel que le prédicat $j \in S$ était faux durant toute l'exécution de GetSet().
  - Éventuellement des js tels que le prédicat $j \in S$ a varié durant l'exécution de GetSet().

Une implémentation adaptative sans-attente d'un objet ensemble actif est décrite dans [8]. *Adaptatif* signifie que le coût de chaque opération (mesuré par le nombre d'accès à la mémoire partagée) dépend du nombre de processus qui ont invoqué Join() et qui n'ont pas encore terminé l'invocation du Leave() correspondant.

Registres LL/SC Comme mentionné au début de la section, ces registres seront utilisés uniquement pour obtenir un algorithme amélioré dans lequel ils remplacent les registres atomiques.

Un registre LL/SC est un registre qui fournit aux processus deux opérations atomiques LL() (Linked Load) et SC(v) (Store Conditional où v est une valeur). Considérons un registre LL/SC X. X.LL() renvoie la valeur actuelle de X. Une écriture conditionnelle X.SC(v) invoquée par un processus p_i renvoie $true$ (l'écriture a réussi) ou $false$ (l'écriture a échoué). Son succès dépend du fait que d'autres processus aient ou non écrit dans X depuis la dernière invocation de X.LL() par p_i. Elle réussit si et seulement si, depuis la dernière lecture de X par p_i (en utilisant X.LL()), X n'a pas été écrit avec succès par un autre processus p_j (quelle que soit la valeur écrite par p_j, celle-ci pouvant être la même que la valeur actuelle de X). Si X.SC(v) réussit, p_i sait donc avec certitude que X n'a pas été écrit depuis sa dernière lecture.

La puissance (du point de vue de la calculabilité) de la paire d'opérations LL/SC en présence de crashes est la même que celle de l'opération Compare&Swap, c'est-à-dire qu'elle a un nombre de consensus égal à $+\infty$ [69].

4.2.2 Définitions

4.2.2.1 Objet snapshot partiel

Un objet snapshot partiel multi-écrivains/multi-lecteurs consiste en m entrées (chacune étant un registre atomique multi-écrivains/multi-lecteurs) qui fournit aux processus deux opérations update() et snapshot() telles que :

- update(r, v, i) est invoquée par p_i pour écrire la valeur v dans l'entrée r ($1 \leq r \leq m$) de l'objet snapshot. Cette opération renvoie la valeur de contrôle ok.

– p_snapshot(R), où R est une séquence $\langle r_1, \cdots, r_x \rangle$ d'indexes d'entrées, permet à un processus d'obtenir la valeur de chaque entrée de R. Elle renvoie une séquence correspondante de valeurs $\langle v_1, \cdots, v_x \rangle$.

Un objet snapshot partiel est défini par les propriétés suivantes.

– Terminaison sans-attente. Chaque invocation update() ou p_snapshot() faite par un processus correct termine.
– Linéarisabilité. Les opérations invoquées par les processus (sauf éventuellement la dernière opération invoquée par un processus fautif [5]) apparaissent comme si elles avaient été exécutées l'une après l'autre, chacune ayant été exécutée à un instant situé entre son premier évènement et son dernier évènement.

Dans le cas du snapshot partiel, la linéarisabilité signifie qu'une opération p_snapshot() renvoie toujours des valeurs qui étaient simultanément présentes en mémoire et qui sont à jour.

4.2.2.2 D'autres propriétés liées à l'implémentation

Ces propriétés, présentées informellement au début de la section, ne sont pas liées à la définition du problème du snapshot partiel mais concernent la manière dont le problème est résolu par les algorithmes implémentant ses opérations.

Comme indiqué précédemment, le but de la localité de l'aide est de réduire les conflits sur la mémoire partagée en empêchant un update$(r, -, -)$ d'aider un p_snapshot(r) concurrent quand $r \notin R$. Plus formellement, soit L_{op} l'ensemble d'entrées accédées par une opération op (notons que L_{op} est un singleton si op est une opération update$(r, -, -)$). Soit M_{op} l'ensemble d'objets de base (registres atomiques et objets ensemble actifs utilisés par l'implémentation) accédés par l'opération op.

Définition 4.7. *Les algorithmes qui implémentent les opérations* update() *et* p_snapshot() *satisfont la propriété de* localité de l'aide *si, pour chaque paire d'opérations concurrentes* $op1$, $op2$ *qui accèdent à un objet de base commun, (1) elles accèdent à une entrée commune ou (2) il existe une opération* $op3$ *concurrente avec* $op1$ *et* $op2$ *telle que (2.1)* $op1$ *et* $op3$ *accèdent à une entrée commune et (2.1)* $op2$ *et* $op3$ *accèdent à une entrée commune. Plus formellement,* $M_{op1} \cap M_{op2} \neq \emptyset \Rightarrow (L_{op1} \cap L_{op2} \neq \emptyset) \vee (\exists op3 : (L_{op1} \cap L_{op3} \neq \emptyset \wedge L_{op3} \cap L_{op2} \neq \emptyset))$.

Cette propriété liée à l'efficacité vient de l'observation que des opérations update$(r, -, -)$ et p_snapshot(R) qui sont concurrentes et telles que $r \notin R$ sont en fait des opérations indépendantes (comme la lecture d'un registre atomique X et l'écriture concurrente d'un registre atomique $Y \neq X$). Intuitivement, la localité de l'aide implique que l'implémentation n'aide que ce qui est nécessaire et suffisant.

Définition 4.8. *Les algorithmes qui implémentent les opérations* update() *et* p_snapshot() *satisfont la propriété de* fraicheur *si une opération* update$(r, v, -)$ *apparait toujours comme si elle avait été exécutée à l'instant de son premier accès à la mémoire partagée (c'est-à-dire que l'update est toujours linéarisé à son premier accès à la mémoire partagée).*

Le but de cette propriété est de fournir aux opérations de snapshot partiel des valeurs "aussi fraiches que possible" avec la signification suivante. Si update$(r, v, -)$ aide p_snapshot(R) (on a donc $r \in R$), la valeur renvoyée pour r est au moins aussi récente que v.

4.2.3 Construction efficace d'un objet snapshot partiel

Cette section présente une construction (figures 4.5 et 4.6) d'un objet de snapshot partiel qui satisfait les propriétés de localité de l'aide et de fraicheur définies précédemment.

5. Si une telle opération n'apparait pas dans la séquence, c'est comme si elle n'avait pas été invoquée.

4.2.3.1 Objets de base

Les algorithmes qui implémentent les opérations p_snapshot() et update() utilisent les variables partagées suivantes.

- Un tableau, noté $REG[1..m]$, de registres atomiques multi-écrivains/multi-lecteurs. Le registre $REG[r]$ est associé à l'entrée r de l'objet snapshot. Il est composé de trois champs $\langle value, pid, sn\rangle$, dont les rôles sont les suivants. Le champ $REG[r].value$ contient la valeur courante de l'entrée r ; $REG[r].pid$ et $REG[r].sn$ sont des valeurs de contrôle associées à cette valeur. $REG[r].pid$ contient l'identité du processus qui a invoqué l'opération update() correspondante et $REG[r].sn$ contient son numéro de séquence parmi toutes les opérations update() invoquées par ce processus.
- Un tableau, noté $AS[1..m]$, d'objets ensemble actif. L'objet $AS[r]$, associé à l'entrée r de l'objet snapshot, contient les identités des processus qui sont en train d'effectuer une opération snapshot() sur r.
- Un tableau, noté $ANNOUNCE[1..n]$, de registres atomiques mono-écrivain/multi-lecteurs. Le registre $ANNOUNCE[i]$ ne peut être écrit que par p_i. Cela a lieu quand p_i invoque p_snapshot(R) : il stocke alors R dans $ANNOUNCE[i]$ (les indexes $r_1, \cdots, r_x$ des entrées qu'il veut lire). De cette manière, si un processus p_j doit aider p_i à terminer son opération p_snapshot(), il n'a qu'à lire $ANNOUNCE[i]$ pour connaitre les entrées qui intéressent p_i.
- Un tableau, noté $HELPSNAP[1..n, 1..n]$, de registres atomiques mono-écrivain/multi-lecteurs. Les registres $HELPSNAP[i, j]$, $1 \leq j \leq n$, ne peuvent être écrits que par p_i. Quand, durant l'exécution d'une opération update(), p_i doit aider p_j à terminer son opération p_snapshot($\langle r_1, \ldots, r_x\rangle$) concurrente, il dépose dans $HELPSNAP[i, j]$ une séquence de valeurs $\langle v_1, \cdots, v_x\rangle$ qui peuvent être utilisées par p_j comme le résultat de son opération p_snapshot($\langle r_1, \cdots, r_x\rangle$).

4.2.3.2 L'opération p_snapshot()

L'algorithme qui implémente cette opération est présenté dans la figure 4.5. Comme dans [22], il emprunte son principe de base à [1]. Plus précisément, il utilise d'abord un "double scan séquentiel" pour essayer de terminer sans aide. Si il n'y arrive pas, il cherche un processus qui pourrait l'aider à terminer (c'est-à-dire un processus qui a effectué deux updates sur une entrée qu'il veut lire).

```
operation p_snapshot(⟨r_1, ··· , r_x⟩) :    % (code for p_i) %
(01) ANNOUNCE[i] ← ⟨r_1, ··· , r_x⟩ ;
(02) can_help_me_i ← ∅ ; for each r ∈ {r_1, ··· , r_x} do AS[r].Join() end for ;
(03) for each r ∈ {r_1, ··· , r_x} do aa[r] ← REG[r] end for ;
(04) while true do % Lin point if return at line 08 %
(05)    for each r ∈ {r_1, ··· , r_x} do bb[r] ← REG[r] end for ;
(06)    if (∀r ∈ {r_1, ··· , r_x} : aa[r] = bb[r]) then
(07)       for each r ∈ {r_1, ··· , r_x} do AS[r].Leave() end for ;
(08)       return(⟨bb[r_1].value, ··· , bb[r_x].value⟩)
(09)    end if ;
(10)    for each r ∈ {r_1, ··· , r_x} such that (aa[r] ≠ bb[r]) do
(11)       can_help_me_i ← can_help_me_i ∪ {⟨bb[r].pid, bb[r].sn⟩}
(12)    end for ;
(13)    if (∃⟨w, sn1⟩, ⟨w, sn2⟩ ∈ can_help_me_i such that sn1 ≠ sn2) then
(14)       for each r ∈ {r_1, ··· , r_x} do AS[r].Leave() end for ;
(15)       return(HELPSNAP[w, i])
(16)    end if ;
(17)    aa ← bb
(18) end while.
```

FIGURE 4.5 – Un algorithme pour l'opération p_snapshot()

Départ Quand il invoque p_snapshot(R), un processus p_i commence par annoncer les entrées qu'il veut lire (ligne 01) et il invoque $AS[r]$.Join() pour chaque $r \in R$ (ligne 02). Cela permet aux processus qui effectuent un update concurrent sur une entrée de R de l'aider.

Double scan séquentiel Le processus p_i entre ensuite dans une boucle (lignes 04-18). Durant chaque exécution de la boucle, il utilise une paire de scans des registres $REG[r]$ pour les entrées qui l'intéressent, c'est-à-dire $\{r_1, \ldots, r_x\}$. Il est important de remarquer que ces scans sont séquentiels [1] : le deuxième scan commence toujours après que le premier ait terminé. Les valeurs lues durant le premier scan sont stockées dans le tableau aa (ligne 03 pour la première boucle, puis ligne 05 suivie de la ligne 17), tandis que les valeurs lues durant le deuxième scan sont stockées dans le tableau bb (ligne 05).

Essayer d'abord de terminer sans aide Si, pour chaque $r \in \{r_1, \ldots, r_x\}$, il n'observe pas de changement dans $REG[r]$ (test de la ligne 06), p_i peut conclure qu'entre la fin du premier scan et le début du deuxième, aucun $REG[r]$, $r \in \{r_1, \ldots, r_x\}$ n'a été modifié. On appelle cela un *double scan réussi*. Les valeurs présentes dans bb étaient donc présentes simultanément dans l'objet snapshot : elles peuvent être renvoyées comme résultats de l'invocation p_snapshot($\langle r_1, \cdots, r_x \rangle$) (ligne 08). Dans ce cas, avant de terminer, p_i invoque $AS[r]$.Leave() pour chaque $r \in R$ pour annoncer qu'il n'a plus besoin d'aide (ligne 07).

Sinon, essayer de bénéficier du mécanisme d'aide Jusque là, le comportement de l'algorithme est le même que ceux de [1, 22]. Il diffère à partir de maintenant. La différence principale est due à l'utilisation de la stratégie "écrire d'abord, aider ensuite" et à son impact sur le fonctionnement de l'algorithme.

Si le test à la ligne 06 n'est pas satisfait, p_i utilise le mécanisme d'aide qui fonctionne de la manière suivante (du coté de p_i). Comme le test est faux, il y a au moins une entrée $r \in \{r_1, \cdots, r_x\}$ qui a été modifiée entre les deux scans. Pour chaque entrée r qui a été modifiée, p_i considère l'identité de la dernière écriture, c'est-à-dire la paire $\langle w, sn \rangle$ extraite de $bb[r] = \langle -, w, sn \rangle$ (le dernier écrivain de $REG[r]$ est p_w et sn est le numéro de séquence associé à l'update correspondant) ; p_i ajoute cette paire à un ensemble local $can\_help\_me_i$ où il note les processus susceptibles de l'aider (lignes 10-12).

p_i vérifie ensuite si il peut terminer grâce au mécanisme d'aide (lignes 13-16). Le prédicat de terminaison est le suivant : "p_i a observé un processus p_w qui a effectué deux updates (sur n'importe quelles entrées)". D'un point de vue opérationnel, ceci est capturé par le fait que p_w apparait deux fois dans $can\_help\_me_i$ (ligne 13). Comme nous le verrons dans la démonstration, le fait que ce prédicat est vrai signifie que p_w a déterminé un ensemble de valeurs $\langle v_1, \cdots, v_x \rangle$ (stockées dans $HELPSNAP[w, i]$) que p_i peut utiliser comme résultat de son opération p_snapshot($\langle r_1, \cdots, r_x \rangle$). Dans ce cas, p_i invoque $AS[r]$.Leave() pour chaque $r \in R$ pour indiquer qu'il n'a plus besoin d'aide et renvoie le contenu de $HELPSNAP[w, i]$ (lignes 14-15).

Finalement, si le prédicat est faux, p_i ne peut pas terminer et recommence donc la boucle (après avoir copié le tableau bb dans aa, ligne 17).

4.2.3.3 L'opération update()

L'algorithme pour l'opération update() est décrit dans la figure 4.6. Le processus p_i qui invoque l'opération écrit d'abord la nouvelle valeur (ligne 01, où nbw_i est un générateur local de numéro de séquence), puis aide les autres processus de manière asynchrone (lignes 02-30). Comme indiqué précédemment, les principes sur lesquels se base ce mécanisme d'aide diffèrent des algorithmes de snapshot préexistants. Soit update(r, v, i) une invocation d'update. Le mécanisme d'aide fonctionne de la manière suivante.

```
operation update(r, v, i) :    % (code for p_i) %
(01) nbw_i ← nbw_i + 1 ; REG[r] ← ⟨v, i, nbw_i⟩ ; % Lin Point %
(02) readers_i ← AS[r].GetSet() ; to_help_i ← ∅ ;
(03) for each j ∈ readers_i do
(04)     announce_i[j] ← ANNOUNCE[j] ;
(05)     if (r ∈ announce_i[j]) then to_help_i ← to_help_i ∪ {j} end if
(06) end for ;
(07) if (to_help_i = ∅) then return(ok) end if ;
(08) to_read_i ← (⋃_{j∈to_help_i} announce_i[j]) expressed as a sequence ⟨rr_1, ..., rr_y⟩ ;
(09) for each j ∈ to_help_i do can_help_i[j] ← ∅ end for ;
(10) for each rr ∈ to_read_i do aa[rr] ← REG[rr] end for ;
(11) while (to_help_i ≠ ∅) do
(12)   for each rr ∈ to_read_i do bb[rr] ← REG[rr] end for ;
(13)   still_to_help_i ← ∅ ;
(14)   for each rr ∈ to_read_i such that aa[rr] ≠ bb[rr] do
(15)     for each j ∈ to_help_i such that rr ∈ announce_i[j] do
(16)       still_to_help_i ← still_to_help_i ∪ {j} ;
(17)       can_help_i[j] ← can_help_i[j] ∪ {⟨bb[r].pid, ⟨bb[r].sn⟩}
(18)     end for
(19)   end for ;
(20)   for each j ∈ to_help_i \ still_to_help_i do
(21)     HELPSNAP[i, j] ← ⟨bb[r_1].value, ..., bb[r_x].value⟩ where ⟨r_1, ..., r_x⟩ = announce_i[j]
(22)   end for ;
(23)   for each j ∈ still_to_help_i do
(24)     if (∃⟨w, sn1⟩, ⟨w, sn2⟩ ∈ can_help_i[j] such that sn1 ≠ sn2) then
(25)       HELPSNAP[i, j] ← HELPSNAP[w, j] ; still_to_help_i ← still_to_help_i \ {j}
(26)     end if
(27)   end for ;
(28)   to_help_i ← still_to_help_i ; to_read_i ← (⋃_{j∈to_help_i} announce_i[j]) ; aa ← bb
(29) end while ;
(30) return(ok).
```

FIGURE 4.6 – Un algorithme pour l'opération update()

Y a-t-il des processus à aider ? Le processus p_i invoque d'abord $AS[r]$.GetSet() pour connaitre l'ensemble de processus qui ont invoqué une opération p_snapshot(R) concurrente telle que $r \in R$ (ligne 02). Il calcule alors l'ensemble to_help_i de processus à aider. (Remarquons qu'un processus p_k renvoyé par $AS[r]$.GetSet() peut avoir terminé son opération de snapshot courante et en avoir démarrer une autre entre les exécutions des lignes 02 et 04 par p_i. Si p_i aidait cet autre snapshot, il violerait la propriété de localité de l'aide. Il utilise le tableau $ANNOUNCE$ pour exclure p_k (lignes 03-06).) Si il n'y a pas de processus en conflit, ($to\_help_i = \emptyset$), p_i n'a pas à aider (propriété de localité de l'aide) et termine (ligne 07).

Si $to\_help_i \neq \emptyset$, p_i peut avoir à aider les processus présents dans to_help_i. Pour cela, il calcule d'abord l'ensemble to_read_i des entrées qu'il doit lire pour aider ces processus (ligne 08). Il initialise aussi un tableau local can_help_i à $\emptyset$ (ligne 09). L'entrée $can\_help_i[j]$ contient les processus p_w qui (à la connaissance de p_i) pourraient aussi aider le processus p_j.

Comment un processus aide individuellement un autre processus Chaque processus p_j de to_help_i est aidé individuellement par p_i. C'est fait dans la boucle (lignes 11-29) qui termine quand l'ensemble to_help_i devient vide.

Dans chaque itération de la boucle, de manière similaire à ce qui est fait dans l'opération p_snapshot(), p_i exécute d'abord un double scan (dont les valeurs sont stockées dans les tableaux locaux aa et bb) et agit de la manière suivante.

– Partie 1 : lignes 13-19. Pour chaque entrée rr telle que $aa[rr] \neq bb[rr]$, soit $bb[rr] = \langle -, w, sn \rangle$, ce qui signifie que (à la connaissance de p_i) p_w est le dernier processus qui a écrit dans l'entrée rr (lignes 14 et 17). De plus, cette écriture a eu lieu durant le double scan. Selon cette observation, p_i note le fait qu'un tel processus p_w pourrait aider chaque p_j tel que $rr \in announce_i[j]$. Ceci est fait en ajoutant la paire $\langle w, sn \rangle$ à l'ensemble $can\_help_i[j]$ (lignes 15-17). De plus, p_i ajoute j à l'ensemble $still\_to\_help_i$ (lignes 13 et 16).
– Partie 2 : lignes 20-22. p_i cherche ensuite les processus qu'il peut aider directement. Ce sont les processus p_j tels que $j \in to\_help_i \setminus still\_to\_help_i$. Les entrées rr qu'ils veulent lire sont telles que $aa[rr] = bb[rr]$, ce qui signifie que la paire (aa, bb) constitue un double scan réussi. p_i écrit alors dans $HELPSNAP[i, j]$ une valeur de snapshot que p_j peut utiliser si son opération de snapshot partiel est toujours en cours. Cette valeur d'aide est $\langle bb[r_1].value, \ldots, bb[r_x].value \rangle$ où $\langle r_1, \ldots, r_x \rangle = announce_i[j]$ (ligne 21).
– Partie 3 : lignes 23-27. Pour chaque processus p_j qui n'a pas été aidé précédemment (ligne 23), p_i cherche si il existe un processus p_w qui peut aider p_j. Le prédicat de terminaison de l'aide (ligne 24) est le même que celui utilisé dans l'algorithme de p_snapshot() (ligne 13 de la figure 4.5) : il y a deux écritures par un processus p_w qui apparaissent dans $can\_help_i[j]$. Si le prédicat est vrai, la valeur d'aide fournie à p_j par p_w est empruntée par p_i pour aider p_j (ligne 25).
– Partie 4 : ligne 28. Finalement, p_i met à jour to_help_i et to_read_i avant de ré-exécuter la boucle. Si $to\_help_i = \emptyset$, la boucle termine.

4.2.4 Démonstration de l'algorithme

La démonstration utilise de manière implicite l'hypothèse classique que l'ensemble des numéros de séquence est aussi grand que nécessaire afin d'éviter d'avoir à réutiliser un numéro de séquence.

Le but des définitions suivantes est d'aider à démontrer que les valeurs renvoyées sont "cohérentes", c'est-à-dire qu'elles viennent des entrées appropriées, qu'elles étaient présentes simultanément dans la mémoire et qu'elles sont récentes [6].

4.2.4.1 Définitions préliminaires

Définition 4.9. *Les valeurs $\langle v_1, \cdots, v_y \rangle$ renvoyées par une opération* p_snapshot($\langle r_1, \ldots, r_x \rangle$) *sont* bien définies *si $x = y$ et pour chaque ℓ, $1 \leq \ell \leq x$, la valeur v_ℓ a été lue dans $REG[r_\ell]$.*

Définition 4.10. *Les valeurs renvoyées par une opération* p_snapshot($\langle r_1, \ldots, r_x \rangle$) *sont* mutuellement cohérentes *si il y a un instant auquel elles étaient présentes simultanément dans l'objet snapshot.*

Définition 4.11. *Les valeurs renvoyées par une opération* p_snapshot($\langle r_1, \ldots, r_x \rangle$) *sont* fraiches *si, pour chaque ℓ, $1 \leq \ell \leq x$, la valeur v_ℓ renvoyée pour r_ℓ n'est pas moins récente que la dernière valeur écrite dans $REG[r_\ell]$ avant l'invocation du snapshot partiel* [7].

Définition 4.12. *Si un processus termine une opération de snapshot partiel à la ligne 08 (figure 4.5), ou exécute la ligne 21 d'une opération d'update (figure 4.6), la dernière paire de lectures du tableau REG constitue un* double scan réussi. *(Le premier scan est la lecture dont les valeurs sont stockées dans le tableau $aa[\,]$, tandis que la deuxième scan est la lecture dont les valeurs sont stockées dans le tableau $bb[\,]$.)*

6. Toujours renvoyer les valeurs initiales fournirait des valeurs bien définies et mutuellement cohérentes, mais elles ne seraient pas récentes et les opérations ne seraient pas linéarisables.

7. Rappelons que, comme chaque $REG[r_\ell]$ est un registre atomique, ses opérations de lecture et d'écriture peuvent être ordonnées totalement. L'expression "dernière valeur écrite" est utilisée par rapport à cet ordre.

Définition 4.13. *Soit* psp *une opération de snapshot partiel invoquée par un processus p_i. La notion de* distance d'aide *des valeurs renvoyées par* psp *est définie de la manière suivante.*

- *La distance d'aide est* 0 *si ces valeurs ont été calculées par un double scan réussi à la ligne 08 de* psp.
- *La distance d'aide est* 1 *si* psp *renvoie les valeurs stockées dans $HELPSNAP[w1, i]$ (figure 4.5, ligne 15), et ces valeurs ont été écrites dans $HELPSNAP[w1, i]$ par p_{w1} à la ligne 21 de la figure 4.6 (elles viennent donc directement d'un double scan réussi intégré à l'opération d'update invoquée par p_{w1}).*
- *Plus généralement, la distance d'aide est h si* psp *renvoie les valeurs qui sont stockées dans $HELPSNAP[w1, i]$ (figure 4.5, ligne 15), et ces valeurs ont été copiées h fois d'un registre $HELPSNAP[-, -]$ vers un autre entre le double scan réussi qui les a calculé et l'opération* psp *qui les renvoie. Remarquons que la première de ces écritures est due à l'exécution de la ligne 21 par une opération d'update (figure 4.6) et suit un double scan réussi par cette opération. Chacune des autres écritures est due à l'exécution de la ligne 25 par une opération d'update.*

4.2.4.2 Les valeurs renvoyées sont bien définies, mutuellement cohérentes et fraiches

Lemme 4.14. *Si la distance d'aide des valeurs renvoyées par une opération* p_snapshot() *est* 0*, ces valeurs sont bien définies, mutuellement cohérentes et fraiches.*

Démonstration. Soit psp $=$ p_snapshot$(\langle r_1, \ldots, r_x \rangle)$ une opération de snapshot partiel qui renvoie des valeurs dont la distance d'aide est $h = 0$. psp termine donc à la ligne 08 et la valeur renvoyée pour chaque entrée $r \in \{r_1, \ldots, r_x\}$ a été obtenue de $bb[r].value$. Le fait que ces valeurs sont bien définies vient directement du fait que, pour chaque $r \in \{r_1, \cdots, r_x\}$, la valeur contenue dans $bb[r]$ a été obtenue à la ligne 05 dans le registre correspondant $REG[r]$.

Pour la cohérence mutuelle et la fraicheur, observons que la terminaison de l'opération p_snapshot() est due à un double scan réussi. Aucune de ces valeurs n'a été modifiée entre ces deux scans (sinon le prédicat de la ligne 06 n'aurait pas été vérifié). Comme ces scans sont séquentiels (le deuxième commence après que le premier ait terminé), on a donc que, à n'importe quel instant entre ces deux scans, les valeurs qui sont renvoyées étaient présentes simultanément dans la mémoire partagée et, pour chaque v_ℓ, aucune écriture n'a eu lieu dans $REG[r_\ell]$ entre l'écriture de $\langle v_\ell, -, - \rangle$ dans $REG[r_\ell]$ et sa lecture dont la valeur a été stockée dans $bb[r_\ell]$. Les valeurs renvoyées sont donc mutuellement cohérentes et fraiches, ce qui conclut la démonstration du lemme. □

Lemme 4.15. *Si la distance d'aide des valeurs renvoyées par une opération* p_snapshot() *est $h > 0$, ces valeurs sont bien définies, mutuellement cohérentes et fraiches.*

Démonstration. Soit $h \geq 1$ la distance d'aide des valeurs renvoyées par psp $=$ p_snapshot$(\langle r_1, \ldots, r_x \rangle)$ et p_i le processus qui l'invoque. Comme $h \geq 1$, psp renvoie les valeurs stockées dans $HELPSNAP[w1, i]$ (ligne 15). Le prédicat évalué à la ligne 13 est donc vrai : il y a un processus p_{w1} tel qu'il y a deux paires distinctes $\langle w1, sn1 \rangle, \langle w1, sn2 \rangle \in can\_help\_me_i$. Sans perte de généralité, soit $sn1 < sn2$. $w1$ a donc été ajouté deux fois à $can\_help\_me_i$ (ligne 11). La démonstration vient de la suite d'observations suivante. Les instants définis et utilisés dans la démonstration sont illustrés dans la figure 4.7. Sauf indication explicite, les numéros de ligne correspondent à la figure 4.5.

1. Comme $w1$ apparait deux fois dans $can\_help\_me_i$, p_{w1} a invoqué deux opérations d'update distinctes update$(r1, v, w1)$ et update$(r2, v', w1)$, qui ont provoqué deux écritures (respectivement dans $REG[r1]$ et dans $REG[r2]$) telles que $r1, r2 \in \{r_1, \ldots, r_x\}$ (remarquons qu'il est possible que $r1 = r2$).
2. L'addition de la paire $\langle w1, sn1 \rangle$ à $can\_help\_me_i$ (qui est due à l'entrée $r1$) implique qu'il y a deux instants t_1 et t_2 et deux valeurs différentes $aa1$ et $bb1$ tels que :

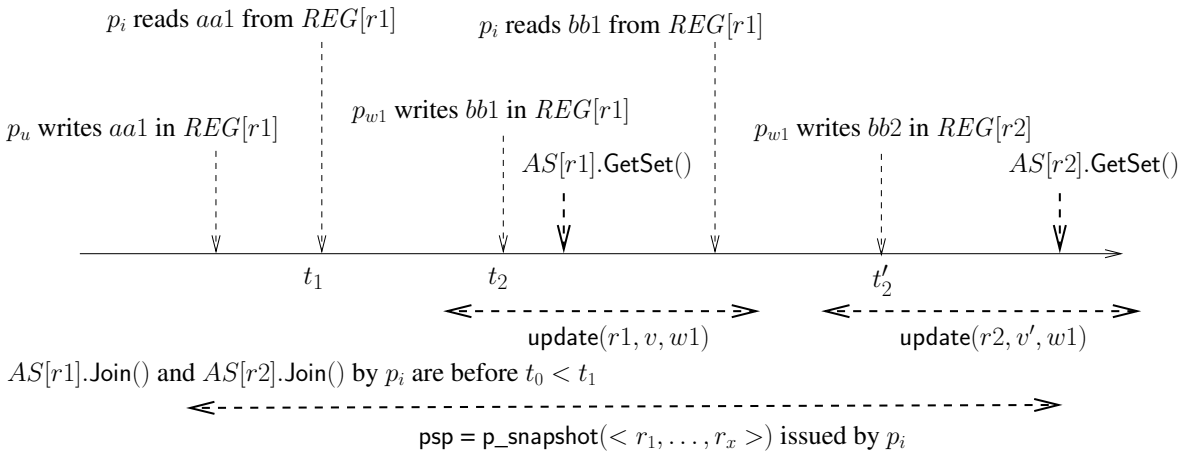

FIGURE 4.7 – Ordre des opérations sur les objets de base

- Il y a un processus p_u qui a écrit $aa1$ dans $REG[r1]$ et cette valeur a ensuite été lue par p_i et stockée dans $aa[r1]$ à l'instant t_1 (ligne 03 ou lignes 05 + 17).
- p_{w1} a écrit $bb1 = \langle v, w1, sn1 \rangle$ dans $REG[r1]$ à l'instant t_2 (ligne 01 de la figure 4.6), et cette valeur a ensuite été lue par p_i et stockée dans $bb[r1]$ (ligne 05).
- Comme p_i a lu dans le registre atomique $REG[r1]$ d'abord $aa1$ puis $bb1 \neq aa1$, on a donc $t_1 < t_2$.

3. De manière similaire, l'addition de $\langle w1, sn2 \rangle$ à $can\_help\_me_i$ (due à $r2$) implique qu'il y a deux instants t'_1 et t'_2 et deux valeurs différentes $aa2$ and $bb2$ tels que :
   - Il y a un processus p_s qui a écrit $aa2$ dans $REG[r2]$, et cette valeur a ensuite été lue par p_i et stockée dans $aa[r2]$ à l'instant t'_1 (ligne 03 ou lignes 05 + 17).
   - p_{w1} a écrit $bb2 = \langle v', w1, sn2 \rangle$ dans $REG[r2]$ à l'instant t'_2 tel que $t'_1 < t'_2$ (ligne 01 de la figure 4.6), et cette valeur a ensuite été lue par p_i et stockée dans $bb[r2]$ (ligne 05).
4. Comme p_{w1} est séquentiel et $sn1 < sn2$, p_{w1} a donc d'abord exécuté update$(r1, v, w1)$ (qui a provoqué l'écriture dans $REG[r1]$) avant d'exécuter update$(r2, v', w1)$ (qui a provoqué l'écriture dans $REG[r2]$). Cette observation implique que $t_2 < t'_2$.
5. Soit t_0 l'instant auquel p_i a démarré son invocation de Join() (ligne 02). Comme p_i exécute Join() avant de lire REG, on a $t_0 < t_1$ et donc $t_0 < t_2 < t'_2$.
6. Quand p_{w1} a exécuté update$(r1, v, w1)$, il a d'abord écrit $REG[r1]$ et a invoqué GetSet() après (lignes 01-02 de la figure 4.6). Comme $t_0 < t_2$, la spécification de l'objet ensemble actif implique que i appartient à l'ensemble $readers^{r1}_{w1}$ renvoyé par l'invocation GetSet() provoquée par update$(r1, v, w1)$.
7. Les valeurs renvoyées par psp ont été déposées dans $HELPSNAP[w1, i]$ par le premier update update$(r1, v, w1)$ ou par le second update update$(r2, v', w1)$. Soit upd cet update. Si elles ont été déposées par le second update, les lignes 02-06 de la figure 4.6 impliquent qu'on a nécessairement $i \in readers^{r2}_{w1}$.
8. Les faits suivants :
   - upd est sur une entrée $r1$ (ou $r2$) que p_i veut lire (point 1),
   - Le prédicat $i \in readers_{w1}$ est vrai quand il est évalué dans upd,
   - p_i met à jour $ANNOUNCE[i]$ à $\langle r_1, \ldots, r_x \rangle$ avant d'invoquer Join(),
   - Join() est invoqué par p_i avant que upd n'invoque GetSet(),
   - upd invoque GetSet() (et obtient $readers_{w1}$) avant de lire le tableau $ANNOUNCE$,

impliquent que upd (qui est le premier ou le second update par p_{w1}) met à jour $announce_{w1}[i]$ à $\langle r_1, \ldots, r_x \rangle$, inclut i dans to_help_{w1} (lignes 04-05 de la figure 4.6) et est tel que son double scan réussi a eu lieu après que psp a démarré.

Remarquons que le raisonnement précédent est indépendant de la notion de distance d'aide. Pour le reste de la démonstration, nous considérons deux cas.

- Cas $h = 1$. Comme la distance d'aide des valeurs renvoyées par psp est 1, elles viennent de $HELPSNAP[w1, i]$ et elles ont été déposées dans ce registre par upd à la ligne 21 (figure 4.6). Comme elles sont les valeurs des entrées de $announce_{w1}[i]$, elles sont bien définies. De plus, comme elles viennent d'un double scan réussi, elles sont mutuellement cohérentes. Enfin, comme le double scan effectué par upd a démarré après le début de psp (point 8), ces valeurs sont fraiches.
- Cas $h > 1$. Observons que, comme les valeurs renvoyées viennent d'un double scan réussi, celles-ci sont mutuellement cohérentes. Nous devons alors montrer qu'elles sont bien définies et fraiches. Les valeurs renvoyées par psp viennent de $HELPSNAP[w1, i]$. Soit $HELPSNAP[w2, i]$ le registre dans lequel ces valeurs ont été lues par p_{w1} avant qu'il les écrive dans $HELPSNAP[w1, i]$ (ligne 25 de la figure 4.6). Soient upw1 l'opération d'update correspondante (par p_{w1}) et upw2 l'opération d'update (par p_{w2}) qui a écrit ces valeurs dans $HELPSNAP[w2, i]$. Supposons (hypothèse d'induction démontrée précédemment pour les cas de base $h = 0$ et $h = 1$) que le double scan réussi duquel viennent les valeurs présentes dans $HELPSNAP[w2, i]$ a été exécuté après le début de upw2. Nous devons montrer que ces valeurs sont bien définies par rapport à psp (c'est-à-dire qu'elles correspondent aux entrées $\langle r_1, \ldots, r_x \rangle$ lues par psp) et qu'elles sont fraiches par rapport à psp (c'est-à-dire que le double scan réussi qui a produit ces valeurs a commencé après le début de psp).
  - Quand on considère les opérations psp et upw1, le point 8 (dans lequel upd est upw1) implique que upw1 a démarré après psp, $i \in readers_{w1}$ et $announce_{w1}[i] = \langle r_1, \ldots, r_x \rangle$.
  - *Affirmation.* upw2 est tel que $i \in readers_{w2}$ et $announce_{w2}[i] = \langle r_1, \ldots, r_x \rangle$, et le double scan réussi qui a calculé les valeurs présentes dans $HELPSNAP[w2, i]$ a démarré après psp.
    Démonstration de l'affirmation. Comme le processus p_{w1} exécute $HELPSNAP[w1, i] \leftarrow HELPSNAP[w2, i]$ (ligne 25, figure 4.6), il y a deux entrées $r1$ et $r2$ telles que (1) $r1, r2 \in announce_{w1}[i] = \langle r_1, \ldots, r_x \rangle$, (2) p_{w2} a effectué deux updates sur $REG[r1]$ et $REG[r2]$, et (3) un de ces updates est upw2.
    Nous sommes alors dans la même situation que celle décrite dans la figure 4.7 où psp et upd sont remplacées respectivement par upw1 et upw2. Le point 8 implique alors que upw2 a démarré après le début de upw1, qui elle-même a démarré après l'invocation de Join() par psp. Donc, (a) upw2 a démarré après psp et (b) l'ensemble renvoyé par l'invocation GetSet() par upw2 inclut i, d'où on conclut que $i \in readers_{w2}$ et $announce_{w2}[i] = \langle r_1, \ldots, r_x \rangle$. Le point (a) et l'hypothèse d'induction que le double scan réussi duquel viennent les valeurs présentes dans $HELPSNAP[w2, i]$ a été exécuté après que upw2 ait commencé impliquent donc que ce double scan réussi a commencé après que psp a démarré. *Fin de la démonstration de l'affirmation.*
  - Les points précédents impliquent que les valeurs présentes dans $HELPSNAP[w1, i]$ renvoyées par psp sont bien définies et fraiches.

□

4.2.4.3 Terminaison sans-attente

Le lemme suivant montre que les algorithmes qui construisent l'objet snapshot partiel sont sans-attente, c'est-à-dire qu'ils terminent malgré le crash d'un nombre quelconque de processus.

Lemme 4.16. *Quand elles sont exécutées par un processus correct, toutes les opérations* update() *et* p_snapshot() *terminent.*

Démonstration. Rappelons que nous supposons une implémentation sans-attente des objets de base ensemble actif. La démonstration est similaire à celle de [1] pour un objet snapshot dont les entrées sont mono-écrivain/multi-lecteurs.

Considérons d'abord une invocation p_snapshot() par un processus correct p_i. Si quand p_i exécute la ligne 06, le prédicat est vrai, l'opération p_snapshot() termine. Supposons donc que ce prédicat n'est jamais satisfait. Nous devons montrer que le prédicat de la ligne 13 devient vrai. Comme le prédicat de la ligne 06 n'est jamais satisfait, chaque fois que p_i exécute la boucle, il y a une entrée r telle que $aa[r] \neq bb[r]$. Le processus p_k qui a modifié $REG[k]$ entre les deux lectures par p_i provoque l'addition de la paire $\langle k, snk \rangle$ à $can\_help\_me_i$ (où $\langle k, snk \rangle$ est extraite de $bb[r]$). Dans le pire cas, $n-1$ paires (chacune associée à un processus, sauf p_i car il ne peut pas exécuter une opération d'update durant son opération de snapshot) peuvent être ajoutées à $can\_help\_me_i$ alors que le prédicat de la ligne 13 reste faux. Mais quand $can\_help\_me_i$ contient une paire par processus (sauf p_i), la prochaine paire qui est ajoutée vient forcément d'un processus p_w tel que $can\_help\_me_i$ contient déjà une paire $\langle w, sn1 \rangle$. Par conséquent, après que la ligne 11 a été exécutée à cause du processus p_w, une deuxième paire $\langle w, sn2 \rangle$ est ajoutée à $can\_help\_me_i$. Le prédicat de la ligne 13 devient alors satisfait, ce qui conclut la démonstration du lemme.

Considérons maintenant une invocation update() par un processus correct p_i. Si, quand il est calculé à la ligne 05, l'ensemble to_help_i reste vide, p_i termine à la ligne 07. Supposons donc que $to\_help_i \neq \emptyset$ et que p_i exécute la boucle **while** (lignes 11-29).

Nous devons montrer que l'ensemble to_help_i devient vide. Les processus qui ne sont pas ajoutés à l'ensemble $still\_to\_help_i$ (ligne 16) sont supprimés de to_help_i (ligne 28). Supposons donc (par contradiction) qu'il y a un ensemble non vide $to\_help\_forever_i \subseteq to\_help_i$ de processus qui sont ajoutés à $still\_to\_help_i$ chaque fois que p_i exécute la boucle **while** (line 16). Soit p_j un de ces processus. Comme $still\_to\_help_i$ est réinitialisé à $\emptyset$ chaque fois que p_i exécute la boucle et p_j reste indéfiniment dans $to\_help\_forever_i$, il existe une paire $\langle w, sn \rangle$ qui est ajoutée à $can\_help_i[j]$ à chaque exécution de la boucle. Le raisonnement est alors le même que pour la terminaison de l'opération p_snapshot(). Une nouvelle paire est ajoutée à $can\_help_i[j]$ à chaque fois que p_i exécute la boucle et il y a $n-1$ processus différents de p_i. Par conséquent, après au plus n exécutions de la boucle, la nouvelle paire $\langle w', sn' \rangle$ qui est ajoutée à $can\_help_i[j]$ est telle que $w = w'$ et $sn \neq sn'$. Ensuite, le prédicat de la ligne 24 devient satisfait et p_j est supprimé de $still\_to\_help_i$, ce qui contredit l'hypothèse initiale et conclut la démonstration du lemme. □

4.2.4.4 Linéarisabilité

Comme mentionné précédemment, la propriété de cohérence d'un objet snapshot est la linéarisabilité. Cela signifie que toutes les invocations des opérations update() et p_snapshot() par les processus durant une exécution (sauf peut-être la dernière invocation par un processus fautif) doivent apparaitre comme si elles avaient été exécutées l'une après l'autre, chacune ayant été exécutée à un instant entre son premier évènement et son dernier évènement.

Lemme 4.17. *Chaque exécution d'un objet snapshot partiel dont les opérations* update() *et* p_snapshot() *sont implémentées avec les algorithmes décrits dans les figures 4.5 et 4.6 est linéarisable.*

Démonstration. La démonstration consiste à associer à chaque opération op un instant (noté $lp(\mathsf{op})$ et appelé son *point de linéarisation*) tel que

- $lp(\mathsf{op})$ est placé entre le début (c'est-à-dire le premier évènement) de op et sa fin (son dernier évènement),
- deux opérations ne partagent pas le même point de linéarisation et
- la séquence des opérations définie par leurs points de linéarisation est une exécution séquentielle de l'objet snapshot.

La démonstration consiste donc en une définition appropriée des points de linéarisation. Le point de linéarisation de chaque opération (sauf peut-être la dernière opération d'un processus fautif) est défini de la façon suivante :

- Une opération update$(r, -, -)$ est linéarisée à l'instant de son écriture dans $REG[r]$ (ligne 01).
- La linéarisation d'une opération snapshot psp = p_snapshot$(\langle r_1, \cdots, r_x \rangle)$ dépend de la ligne à laquelle son instruction return() est exécutée.
  - Cas 1 : psp termine à la ligne 08 grâce à un double scan réussi (la distance d'aide est 0). Son point de linéarisation est placé à n'importe quel instant entre son premier scan et son deuxième scan de ce double scan réussi.
  - Cas 2 : psp termine à la ligne 15 (la distance d'aide est h avec $h > 0$). Dans ce cas, les valeurs renvoyées par psp ont été calculées par une opération update à la ligne 21 ou à la ligne 25. De plus, dans les deux cas, elles ont été calculées par un double scan réussi par un processus p_z. En considérant ce double scan réussi, lp(psp) est alors placé entre la fin du premier scan et le début du deuxième.

La définition précédente des points de linéarisation implique que chaque opération est linéarisée entre son début et sa fin et que deux opérations ne partagent pas le même point de linéarisation [8]. De plus, les lemmes 4.14 et 4.15 concernant la bonne définition, la cohérence mutuelle et la fraicheur des valeurs renvoyées impliquent que les opérations d'update et de snapshot apparaissent comme si elles avaient été exécutées selon l'ordre total défini par leurs points de linéarisation, ce qui conclut la démonstration du lemme. □

4.2.4.5 Fraicheur et localité de l'aide

Lemme 4.18. *Les algorithmes des opérations* update() *et* p_snapshot() *satisfont la fraicheur et la localité de l'aide.*

Démonstration. L'objet ensemble actif $AS[r]$ n'est accédé que par des processus qui exécutent des opérations sur r. Les registres $ANNOUNCE[i]$ et $HELPSNAP[i]$ ne sont accédés que par p_i durant son exécution d'une opération snapshot() et par d'autres processus durant des opérations update() qui sont en conflit avec une opération snapshot() invoquée par p_i. Les seuls objets de base restants qui pourraient invalider la localité de l'aide sont donc les registres de base $REG[1..m]$.

Une opération snapshot() n'accède qu'aux registres correspondant aux entrées qu'il veut lire (lignes 03 et 06). Un snapshot() ne peut donc lire les mêmes objets de base qu'un autre snapshot() que si les ensembles qu'ils lisent s'intersectent. Une opération update() ne lit que les registres de base que des opérations $snapshot()$ concurrentes et en conflit essayent de lire. Le seul cas restant est celui où deux opérations update() lisent les mêmes registres ; cela ne peut arriver que si elles partagent une opération $snapshot()$ concurrente et en conflit. Les algorithmes respectent donc la propriété de localité de l'aide.

La définition des points de linéarisation dans le lemme 4.17 implique la propriété de fraicheur. □

4.2.4.6 L'objet snapshot partiel est correct, sans-attente et efficace

Théorème 4.19. *Les algorithmes décrits dans les figures 4.5 et 4.6 satisfont les propriétés de terminaison et de cohérence (présentées dans la section 4.2.2.1) qui définissent un objet snapshot partiel. De plus, ils satisfont les propriétés de fraicheur et de localité de l'aide.*

Démonstration. La démonstration est la conséquence immédiate des lemmes 4.16, 4.17 et 4.18. □

8. Si deux opérations sont linéarisées au même point, on peut les arbitrer en utilisant les identités des processus qui ont invoqué ces opérations.

4.2.5 Utilisation de registres LL/SC

Un tableau de registres LL/SC Cette section montre qu'en utilisant des registres LL/SC au lieu de registres atomiques classiques comme objets de base, le nombre d'objets utilisés peut être réduit de $O(n^2)$ à $O(n)$. Plus précisément, le tableau $ANNOUNCE[1..n]$ et la matrice $HELPSNAP[1..n, 1..n]$, tous les deux constitués de registres atomiques, peuvent être remplacés par un seul tableau $ANNHELP[1..n]$ de registres offrant les opérations LL/SC.

Le tableau $ANNHELP[1..n]$ Le registre $ANNHELP[i]$ est utilisé à la fois par p_i et par un autre processus $p_j \neq p_i$ pour faire passer de l'information (dans les deux sens). Il peut contenir trois types de valeurs décrits ci-dessous.

- Quand il invoque p_snapshot(R), le processus p_i écrit $\langle req, R\rangle$ dans $ANNHELP[i]$ pour annoncer qu'il veut lire de manière atomique les entrées de R.
- Quand il termine son opération p_snapshot(R) sans avoir été aidé par un autre processus (double scan réussi), p_i écrit $\perp$ dans $ANNHELP[i]$ pour empêcher d'autres processus de l'aider plus tard.
- Quand il aide p_i, un processus p_j écrit dans $ANNHELP[i]$ les valeurs correspondant à l'ensemble d'entrées R que p_i veut lire.

$\langle req, R\rangle$, $\perp$ et $\langle v_1, \ldots, v_x\rangle$ sont donc les trois types de valeurs que $ANNHELP[i]$ peut contenir. Sa valeur initiale est $\perp$.

L'opération update() **basée sur des registres LL/SC** L'opération update() est décrite dans la figure 4.8. Elle est la même que celle de la figure 4.6 sauf 4 lignes qui ont été modifiées (leur numéro termine par "M"). Plus précisément, les modifications sont les suivantes. Soit p_i le processus qui exécute update(r, v, i).

- Lignes 04.M et 05.M. Le processus p_i invoque maintenant $ANNHELP[j]$.LL() pour apprendre l'annonce de chaque lecteur concurrent p_j. Si $ANNHELP[j]$.LL() $\neq \langle req, -\rangle$, p_i n'a pas à aider p_j car (a) p_j a déjà été aidé par un autre processus p_k (dans ce cas $ANNHELP[j]$ contient une liste de valeurs écrites par p_k à la ligne 21.M) ou (b) p_j a terminé son invocation p_snapshot(R) sans être aidé (dans ce cas, comme nous le verrons dans l'algorithme de l'opération p_snapshot(), $ANNHELP[j] = \perp$).
- Ligne 21.M. Comme indiqué au point précédent, cette ligne est exécutée quand p_j aide p_i. Pour cela, p_i exécute $ANNHELP[j]$.SC$(\langle bb[r_1].value, \ldots, bb[r_x].value\rangle)$ pour stocker dans $ANNHELP[j]$ l'aide qu'il donne à p_j. Selon la sémantique de LL/SC, l'invocation $ANNHELP[j]$.SC() par p_i à la ligne 21.M ne réussit que si (1) aucun autre processus n'a aidé p_j depuis la dernière invocation $ANNHELP[j]$.LL() par p_i et (2) p_j n'a pas encore terminé son opération de snapshot partiel. Cela garantit que p_i n'écrase pas une nouvelle annonce de p_j.
- Ligne 25.M. Contrairement à l'algorithme basé sur des registres classiques, où l'aide de p_i à p_j est individuelle et est donc écrite dans le registre $HELPSNAP[i, j]$ (ligne 25 de la figure 4.6), l'aide de p_i à p_j est maintenant écrite directement dans $ANNHELP[j]$ à la ligne 21.M. Par conséquent, si le test à la ligne 24 est satisfait, un autre processus p_w a déjà aidé p_j, et p_i n'a donc plus besoin de transmettre cette aide. La ligne 25.M est donc la ligne 25 sans la transmission de l'aide à p_j.

L'opération p_snapshot() **basée sur des registres LL/SC** L'opération p_snapshot(R) est décrite dans la figure 4.8. Elle est la même que celle de la figure 4.5 sauf 2 lignes qui ont été modifiées (leur numéro termine par "M") et une ligne qui a été ajoutée (son numéro termine par "A"). Plus précisément, les modifications sont les suivantes. Soit p_i le processus qui exécute p_snapshot(R). Pour faciliter la compréhension, la ligne 15.M est d'abord expliquée, puis la ligne 07.A et enfin la ligne 01.M.

- Ligne 15.M. Si p_i est aidé par un autre processus p_w, ce processus a déposé les valeurs d'aide pour R dans $ANNHELP[i]$. Par conséquent, p_i renvoie ces valeurs pour les entrées de R.

```
operation update(r, v, i) :    % (code for p_i) %
(01)  nbw_i ← nbw_i + 1 ; REG[r] ← ⟨v, i, nbw_i⟩ ; % Lin Point %
(02)  readers_i ← AS[r].GetSet() ; to_help_i ← ∅ ;
(03)  for each j ∈ readers_i do
(04.M)   annhelp_i[j] ← ANNHELP[j].LL() ;
(05.M)   if (annhelp_i[j] = ⟨req, ann⟩) ∧ (r ∈ ann) then announce_i[j] ← ann ; to_help_i ← to_help_i ∪ {j} end if
(06)  end for ;
(07)  if (to_help_i = ∅) then return(ok) end if ;
(08)  to_read_i ← (⋃_{j∈to_help_i} announce_i[j]) expressed as a sequence ⟨rr_1, ..., rr_y⟩ ;
(09)  for each j ∈ to_help_i do can_help_i[j] ← ∅ end for ;
(10)  for each rr ∈ to_read_i do aa[rr] ← REG[rr] end for ;
(11)  while (to_help_i ≠ ∅) do
(12)     for each rr ∈ to_read_i do bb[rr] ← REG[rr] end for ;
(13)     still_to_help_i ← ∅ ;
(14)     for each rr ∈ to_read_i such that aa[rr] ≠ bb[rr] do
(15)        for each j ∈ to_help_i such that rr ∈ announce_i[j] do
(16)           still_to_help_i ← still_to_help_i ∪ {j} ;
(17)           can_help_i[j] ← can_help_i[j] ∪ {⟨bb[r].pid, ⟨bb[r].sn⟩}
(18)        end for
(19)     end for ;
(20)     for each j ∈ to_help_i \ still_to_help_i do
(21.M)      ANNHELP[j].SC(⟨bb[r_1].value, ..., bb[r_x].value⟩) where ⟨r_1, ..., r_x⟩ = announce_i[j]
(22)     end for ;
(23)     for each j ∈ still_to_help_i do
(24)        if (∃⟨w, sn1⟩, ⟨w, sn2⟩ ∈ can_help_i[j] such that sn1 ≠ sn2) then
(25.M)         still_to_help_i ← still_to_help_i \ {j}
(26)        end if
(27)     end for ;
(28)     to_help_i ← still_to_help_i ; to_read_i ← (⋃_{j∈to_help_i} announce_i[j]) ; aa ← bb
(29)  end while ;
(30)  return(ok).
```

FIGURE 4.8 – Une opération update() basée sur des registres LL/SC

- Ligne 07.A. Si p_i a terminé son opération de snapshot partiel sans l'aide d'un autre processus (double scan réussi), il essaie d'écrire $\perp$ dans $ANNHELP[i]$. Remarquons que cette écriture réussit si aucun processus n'a déposé de valeur d'aide dans $ANNHELP[i]$.
  De cette manière, quand p_i termine son opération de snapshot partiel (aidé à la ligne 15.M ou sans aide à la ligne 08), on a nécessairement $ANNHELP[i] \neq \langle req, -\rangle$. Cela signifie que, entre la fin de cette invocation p_snapshot() par p_i et le début de la prochaine invocation p_snapshot(), aucun processus p_j ne peut l'aider (pour que p_j aide p_i, p_j doit trouver $ANNHELP[i].\mathsf{LL}() = \langle req, -\rangle$, comme décrit à la ligne 05.M de la figure 4.8).
- Ligne 01.M. Quand le processus p_i commence une opération p_snapshot(R), il invoque d'abord $ANNHELP[i].\mathsf{SC}(\langle req, R\rangle)$ pour annoncer les entrées qu'il veut lire. Remarquons que, selon l'observation faite au point précédent, la séquence LL/SC de la ligne 01.M garantit que cette écriture conditionnelle réussit toujours.

4.2.6 Conclusion

Le concept de snapshot en mémoire partagée a été introduit dans [1, 11]. La notion de snapshot partiel a ensuite été introduite dans [22]. Cette section a d'abord proposé deux propriétés liées à l'implémentation d'un objet snapshot partiel. Elle a ensuite présenté une construction d'un objet snapshot partiel dont l'implémentation satisfait ces propriétés. Ses caractéristiques principales sont les suivantes [9].

9. Ces caractéristiques sont aussi les différences principales avec l'algorithme (basé sur des registres) de snapshot partiel présenté dans [22] et les algorithmes (basés sur des registres) qui implémentent un objet snapshot classique (par exemple [1]).

```
operation p_snapshot(⟨r_1, ··· , r_x⟩) :    % (code for p_i) %
(01.M) ANNHELP[i].LL() ; ANNHELP[i].SC(⟨req, ⟨r_1, ··· , r_x⟩⟩) ;
(02)   can_help_me_i ← ∅ ; for each r ∈ {r_1, ··· , r_x} do AS[r].Join() end for ;
(03)   for each r ∈ {r_1, ··· , r_x} do aa[r] ← REG[r] end for ;
(04)   while true do % Lin point if return at line 08 %
(05)      for each r ∈ {r_1, ··· , r_x} do bb[r] ← REG[r] end for ;
(06)      if (∀r ∈ {r_1, ··· , r_x} : aa[r] = bb[r]) then
(07)         for each r ∈ {r_1, ··· , r_x} do AS[r].Leave() end for ;
(07.A)       if ANNHELP[i].LL() = ⟨req, −⟩) then ANNHELP[i].SC(⊥) end if ;
(08)         return(⟨bb[r_1].value, ··· , bb[r_x].value⟩)
(09)      end if ;
(10)      for each r ∈ {r_1, ··· , r_x} such that (aa[r] ≠ bb[r]) do
(11)         can_help_me_i ← can_help_me_i ∪ {⟨bb[r].pid, bb[r].sn⟩}
(12)      end for ;
(13)      if (∃⟨w, sn1⟩, ⟨w, sn2⟩ ∈ can_help_me_i such that sn1 ≠ sn2) then
(14)         for each r ∈ {r_1, ··· , r_x} do AS[r].Leave() end for ;
(15.M)       return(ANNHELP[i])
(16)      end if ;
(17)      aa ← bb
(18)   end while.
```

FIGURE 4.9 – Une opération p_snapshot() basée sur des registres LL/SC

1. L'algorithme proposé est le premier (à notre connaissance) à être basé sur la stratégie "écrire d'abord, aider ensuite".
2. Une opération d'update aide une opération de snapshot si nécessaire, mais pas plus. Ceci est capturé formellement par la propriété de localité de l'aide qui exprime "pas d'aide sans conflit".
3. Les algorithmes d'update proposés précédemment effectuaient une seule écriture en mémoire partagée. L'algorithme présenté ici sépare l'écriture d'une nouvelle valeur des écritures de valeurs d'aide.
4. L'opération d'update satisfait la propriété de *fraicheur* suivante : la valeur écrite par une opération d'update est visible par les autres processus dès le premier accès à la mémoire partagée par cet update.
5. Le nombre de registres de base peut être réduit de $O(n^2)$ à $O(n)$ si ces registres peuvent être accédés par les opérations LL/SC plutôt que par des lectures et écritures classiques.

Chapitre 5

Simulations

Une question fondamentale dans l'étude de l'informatique distribuée est celle de la calculabilité : étant donnée une tâche, dans quels systèmes et sous quelles conditions de progression peut-elle être résolue ? Un travail fondateur dans ce domaine a été réalisé par Borowsky et Gafni : la BG simulation.

Étant donnée une tâche non colorée, la BG simulation montre que cette tâche peut être résolue de manière t-résiliente dans un système de n processus qui communiquent à travers des registres partagés si et seulement si elle peut être résolue de manière sans-attente dans un système de $t+1$ processus (qui n'ont également accès qu'à des registres partagés). Pour arriver à ce résultat, on simule de manière distribuée un système de n processus dans un système de $t+1$ processus. Dans cette simulation, le crash d'un simulateur entraine le crash d'au plus un processus simulé, ce qui permet de résoudre la tâche si au plus t simulateurs crashent.

La simulation BG étendue La simulation BG de base traite le cas des tâches non colorées. En effet, dans cette simulation, rien n'empêche deux processus distincts de choisir la même sortie. Avec une tâche colorée, ce n'est pas possible. Le meilleur exemple est la tâche de renommage, dont la spécification indique explicitement que tous les processus doivent choisir des sorties différentes.

Pour traiter le cas des tâches non colorées, la simulation BG étendue [55] fait correspondre à chaque tâche t-résiliente sur n processus une tâche sans-attente équivalente sur $t+1$ processus (comme pour la simulation classique). La différence réside dans le fait que, dans la simulation étendue, chaque simulateur a un processus simulé qui lui est assigné. Cela permet de traiter toutes les tâches, y compris les tâches non colorées comme le renommage qui ne pouvaient pas être traitées par la simulation BG classique.

Dans [55], un simulateur ignore au départ quel processus lui sera assigné. Nous présentons ici une manière alternative d'implémenter la simulation BG étendue dans laquelle chaque simulateur connait, dès le départ de la simulation, quel processus simulé lui est assigné.

Objets plus puissants que des registres Les simulations BG classique et étendue ne traitent que le cas des systèmes dans lesquels les processus n'ont accès qu'à des registres (sous la forme d'objets snapshot, ce qui est équivalent). Elles n'apportent donc pas de réponse à la question suivante : "étant donné un système dans lequel les processus ont accès à des objets plus puissants que des registres, existe-t-il un système équivalent dans lequel les processus n'ont pas accès à ces objets ?"

Nous présentons une simulation qui montre qu'une tâche non colorée peut être résolue dans un système dans lequel les processus ont accès au x-consensus (consensus parmi n'importe quel sous-ensemble de x processus) et dans lequel jusqu'à t processus peuvent crasher si et seulement si cette tâche peut être résolue dans un système dans lequel les processus n'ont accès qu'à des registres et dans lequel jusqu'à $\lfloor \frac{t}{x} \rfloor$ processus peuvent crasher. Un système dans lequel t processus peuvent crasher et où les processus ont accès au x- consensus est donc équivalent à un système dans lequel t' processus peuvent crasher et où les processus ont accès au x'-consensus si et seulement si $\lfloor \frac{t}{x} \rfloor = \lfloor \frac{t'}{x'} \rfloor$.

5.1 Un algorithme de simulation BG étendue

En plus de l'introduction de la notion de simulation BG étendue, Gafni présente dans [55] un algorithme de simulation BG étendue (appelé ici GeBG). Cet algorithme est basé sur une séquence de sous-protocoles dans laquelle chaque sous-protocole est soit le protocole d'accord utilisé dans la BG classique (le safe_agreement) soit un protocole commit-adopt [52]. Cet algorithme est présenté informellement en anglais.

Cette section présente un autre algorithme de simulation BG étendue particulièrement simple. Il est basé sur deux types d'objets : le safe_agreement (utilisé dans la simulation BG classique et dans GeBG) et un nouveau type d'objet que nous appelons *arbitre* (arbiter). Un objet arbitre permet d'exploiter la notion de "propriétaire" de l'objet : (1) une valeur est toujours décidée quand le processus propriétaire de l'objet ne crashe pas et (2) la valeur de l'objet est déterminée soit par le simulateur propriétaire de l'objet soit par les autres simulateurs.

Comme dans la simulation BG classique, chaque simulateur simule les n processus du système simulé mais, à la différence de celle-ci, chacun des premier $t+1$ processus simulés est associé à exactement un simulateur (son "propriétaire"). L'utilisation d'objets arbitres permet d'assurer que le blocage permanent (crash) d'un de ces premiers $t+1$ processus simulés ne peut être provoqué que par le crash de son propriétaire.

5.1.1 Processus simulés vs processus simulateurs

But Soit A un algorithme t-résilient pour n processus qui résout une tâche de décision dans le modèle asynchrone dans lequel les processus ont accès à des objets snapshot [1]. Le but est de construire un algorithme A' sans attente pour $t+1$ processus qui simule A dans le même modèle.

Notations Un processus simulé est noté p_j avec $1 \leq j \leq n$. De manière similaire, un processus simulateur est noté q_i avec $1 \leq i \leq t+1$. Comme dans le reste de ce document, les objets partagés sont notés en lettre majuscules et les objets locaux en lettres minuscules. La subtilité réside dans le fait qu'un objet partagé est partagé entre les simulateurs et un objet local l'est par rapport à un simulateur.

Rôle d'un simulateur Chaque simulateur q_i reçoit le code de tous les processus simulés $p_1, \ldots, p_n$. Il gère n threads, chacun associé à un processus simulé, et exécute localement ces threads d'une manière équitable. Il gère aussi une copie locale mem_i de la mémoire snapshot mem partagée par les processus simulés.

Le code d'un processus simulé contient des invocations d'écriture dans $mem[j]$ et des invocations de $mem.\mathsf{snapshot}()$. Ce sont les seules opérations utilisées par les processus simulés $p_1, \ldots, p_n$ pour coopérer. Le cœur de la simulation est donc la définition de ces algorithmes. Le premier (noté $\mathsf{sim\_write}_{i,j}()$) doit décrire ce que doit faire un simulateur q_i pour simuler correctement une écriture dans $mem[j]$ invoquée par un processus simulé p_j. Le deuxième (noté $\mathsf{sim\_snapshot}_{i,j}()$) doit décrire ce que doit faire un simulateur q_i pour simuler correctement une invocation de $mem.\mathsf{snapshot}()$ par un processus simulé p_j.

5.1.2 Objets de base utilisés dans la simulation

En plus des objets snapshot, les simulateurs coopèrent en utilisant des registres atomiques et des objets safe_agreement et arbiter dont les spécifications et les implémentations sont décrites dans cette section. Ces objets peuvent être implémentés en utilisant des registres multi-écrivains/multi-lecteurs, qui

1. Comme précisé dans le chapitre 2, ce modèle est équivalent au modèle doté de registres.

eux-mêmes peuvent être implémentés en utilisant des objets snapshot. Le modèle de base reste donc le modèle dans lequel les processus n'ont accès qu'à des snapshots.

5.1.2.1 Le type d'objet safe_agreement

Ce type d'objet (défini dans [27, 30]) est au cœur de la simulation BG. Il fournit à chaque simulateur q_i deux opérations notées $\mathsf{propose}_i(v)$ et $\mathsf{decide}_i()$ que q_i peut invoquer au maximum une fois, dans cet ordre. L'opération $\mathsf{propose}_i(v)$ permet à q_i de proposer une valeur v tandis que $\mathsf{decide}_i()$ lui permet de décider une valeur. Les propriétés satisfaites par un objet du type safe_agreement sont les suivantes.

- *Terminaison*. Si aucun simulateur q_x ne crashe durant son exécution de $\mathsf{propose}_x()$, alors chaque simulateur correct q_i qui invoque $\mathsf{decide}_i()$ termine cette invocation.
- *Accord*. Au plus une valeur est décidée.
- *Validité*. Une valeur décidée est une valeur proposée.

init : **for each** $x : 1 \leq x \leq t+1$ **do** $SM[x] \leftarrow (\perp, 0)$ **end for**.

operation $\mathsf{propose}_i(v)$: % $1 \leq i \leq t+1$ %
(01) $SM[i] \leftarrow (v, 1)$;
(02) $sm_i \leftarrow SM.\mathsf{snapshot}()$;
(03) **if** $(\exists x : sm_i[x].level = 2)$ **then** $SM[i] \leftarrow (v, 0)$ **else** $SM[i] \leftarrow (v, 2)$ **end if**.

operation $\mathsf{decide}_i()$: % $1 \leq i \leq t+1$ %
(04) **repeat** $sm_i \leftarrow SM.\mathsf{snapshot}()$ **until** $(\forall x : sm_i[x].level \neq 1)$ **end repeat** ;
(05) **let** $x = \min(\{k \mid sm_i[k].level = 2)$; $res \leftarrow sm_i[x].value$;
(06) $\mathsf{return}(res)$.

FIGURE 5.1 – Une implémentation du type safe_agreement [30] (code de q_i)

Une implémentation L'implémentation du type safe_agreement décrite dans la figure 5.1 vient de [30]. Cette construction est basée sur un objet snapshot SM (avec une entrée par simulateur q_i). Chaque entrée $SM[i]$ de l'objet snapshot a deux champs : $SM[i].value$ qui contient une valeur et $SM[i].level$ qui contient son niveau. Le niveau 0 signifie que la valeur correspondante est sans importance, 1 signifie qu'elle est instable et 2 qu'elle est stable.

Quand un simulateur q_i invoque $\mathsf{propose}_i(v)$, il écrit d'abord la paire $(v, 1)$ dans $SM[i]$ (ligne 01), puis lit l'objet snapshot SM (ligne 02). S'il y a une valeur stable dans SM, p_i "annule" la valeur qu'il propose, sinon il la rend stable (ligne 03).

Un simulateur q_i invoque $\mathsf{decide}_i()$ après avoir invoqué $\mathsf{propose}_i()$. Son but est de renvoyer la même valeur stable à tous les processus qui invoquent cette opération (ligne 06). Pour cela, q_i calcule en boucle un snapshot de SM jusqu'à ce qu'il n'observe plus de valeur instable dans SM (ligne 04). Remarquons que, comme un simulateur q_i n'invoque $\mathsf{decide}_i()$ qu'après avoir invoqué $\mathsf{propose}_i(v)$, il y a au moins une valeur stable dans SM quand il exécute la ligne 05. Finalement, pour que la même valeur stable soit renvoyée à tous les processus, q_i renvoie la valeur stable proposée par le processus qui a l'identité la plus petite (ligne 05).

Une preuve formelle que cet algorithme implémente le type safe_agreement est donnée dans [30].

5.1.2.2 Le type d'objet arbiter

Définition Chaque objet du type arbiter a un simulateur propriétaire q_j prédéfini statiquement. Un tel objet fournit aux processus une seule opération notée $\mathsf{arbitrate}_{i,j}()$ (où i est l'identité du processus qui l'invoque et j l'identité du propriétaire). Un simulateur q_i invoque $\mathsf{arbitrate}_{i,j}()$ au plus une fois et, quand

elle termine, cette invocation renvoie une valeur à q_i. Les propriétés d'un objet arbiter dont le propriétaire est q_j sont les suivantes.

- *Terminaison*. Si le propriétaire q_j invoque $\mathsf{arbitrate}_{j,j}()$ et est correct, ou s'il n'invoque pas $\mathsf{arbitrate}_{j,j}()$, ou si un simulateur q_i termine son invocation $\mathsf{arbitrate}_{i,j}()$, alors tous les processus corrects terminent leur invocation $\mathsf{arbitrate}_{i,j}()$.
- *Accord*. Au plus une valeur est décidée.
- *Validité*. La valeur renvoyée est 1 (*propriétaire*) ou 0 (*autre*). De plus, si le propriétaire n'invoque pas $\mathsf{arbitrate}_{j,j}()$, la valeur 1 ne peut pas être renvoyée et, si seul le propriétaire invoque $\mathsf{arbitrate}_{i,j}()$, la valeur 0 ne peut pas être renvoyée.

Une implémentation Une implémentation d'un objet du type arbiter est décrite dans la figure 5.2. Elle est basée sur un objet snapshot $PART$ (initialisé à $[false, \ldots, false]$) et un registre atomique $WINNER$ (initialisé à $\perp$).

Quand il invoque $\mathsf{arbitrate}_{i,j}()$, le simulateur q_i annonce qu'il participe (ligne 01) et effectue un snapshot pour connaitre l'ensemble des simulateurs qui ont commencé à participer (line 02). Si q_i est le propriétaire de l'objet, ($i = j$, ligne 03), il vérifie s'il est le premier participant (prédicat $part_i = \{i\}$). Si c'est le cas, il affecte 1 à $WINNER$. Sinon, il lui affecte 0 (line 04). Si p_i n'est pas le propriétaire de l'objet ($i \neq j$), il vérifie si le propriétaire a commencé sa participation (prédicat $j \in part_i$). Si c'est le cas, q_i attend de savoir quelle valeur a été affectée à $WINNER$. Sinon, il affecte 0 à $WINNER$. Finalement, q_i termine en renvoyant la valeur de $WINNER$.

```
operation arbitrate_{i,j}() :   % 1 ≤ i, j ≤ t + 1 %
(01) PART[i] ← true ;
(02) aux_i ← PART.snapshot() ; part_i ← {x | aux_i[x]} ;
(03) if (i = j)   % p_i is the owner of the associated arbiter type object %
(04)    then if (part_i = {i}) then WINNER ← 1 else WINNER ← 0 end if
(05)    else if (j ∈ part_i) then wait (WINNER ≠ ⊥) else WINNER ← 0 end if
(06) end if ;
(07) return(WINNER).
```

FIGURE 5.2 – L'opération $\mathsf{arbitrate}_{i,j}()$ du type arbiter (code de q_i)

La démonstration que cette construction est une implémentation correcte du type arbiter est donnée ci-dessous.

Lemme 5.1. *Si le propriétaire participe et est correct, alors tous les processus corrects qui participent terminent.*

Démonstration. Il n'y a pas de boucle et la seule instruction bloquante de l'algorithme est l'instruction wait à la ligne 05 dans laquelle un processus p_i attend qu'une valeur soit assignée à $WINNER$. Le propriétaire (p_j) affecte une valeur à $WINNER$ avant de terminer (ligne 04), donc si le propriétaire participe et est correct, alors tous les processus corrects qui participent terminent. □

Lemme 5.2. *Si le propriétaire ne participe pas, alors tous les processus corrects qui participent terminent.*

Démonstration. De nouveau, il n'y a pas de boucle et la seule instruction bloquante de l'algorithme est l'instruction wait à la ligne 05. Cette instruction n'est exécutée que si le processus observe que le propriétaire a commencé à participer donc, si le propriétaire ne participe pas, alors tous les processus corrects qui participent terminent. □

Lemme 5.3. *Si un processus termine, alors tous les processus corrects qui participent terminent.*

Démonstration. De nouveau, il n'y a pas de boucle et la seule instruction bloquante de l'algorithme est l'instruction wait à la ligne 05 (elle attend qu'une valeur soit affectée à $WINNER$). Si un processus n'exécute pas cette instruction, il affecte une valeur à $WINNER$. Donc si un processus termine, alors tous les processus corrects qui participent terminent. □

Lemme 5.4. *Tous les processus renvoient la même valeur.*

Démonstration. Le seul processus qui peut affecter à $WINNER$ une valeur différente de 0 est le propriétaire (ligne 04). Nous n'avons donc que ce cas à considérer. Si le propriétaire affecte la valeur 1 à $WINNER$, cela signifie que dans le snapshot qu'il a pris à la ligne 02, il n'a observé aucun autre processus (ligne 04). Comme les snapshots sont linéarisables et le propriétaire annonce qu'il a démarré avant de prendre le snapshot (ligne 01), tous les autres processus observeront que le propriétaire a commencé et exécuteront l'instruction d'attente (ligne 05) au lieu d'affecter la valeur 0 à $WINNER$. Tous les processus renvoient donc la même valeur. □

Lemme 5.5. *Si le propriétaire ne participe pas, la valeur renvoyée est* 0*.*

Démonstration. Le seul processus qui peut affecter une valeur différente de 0 à $WINNER$ est le propriétaire (ligne 04). Donc si le propriétaire ne participe pas, la valeur de $WINNER$ est 0 et tous les processus renvoient cette valeur. □

Lemme 5.6. *Si le propriétaire est le seul participant, la valeur renvoyée est* 1*.*

Démonstration. Si le propriétaire n'observe aucun autre participant, il affecte la valeur 1 à $WINNER$. Donc si le propriétaire est le seul participant, la valeur renvoyée est 1. □

Théorème 5.7. *L'algorithme présenté dans la figure 5.2 est une implémentation correcte du type d'objet* arbiter*.*

Démonstration. Les lemmes 5.1, 5.2, 5.3, 5.4, 5.5 et 5.6 montrent que l'algorithme de la figure 5.2 respecte les spécifications du type d'objet arbiter. □

5.1.3 La simulation BG

Cette section présente la simulation BG classique [27, 30] : ses principes de bases et les algorithmes qui implémentent ses opérations $\mathsf{sim\_write}_{i,j}()$ et $\mathsf{sim\_snapshot}_{i,j}()$.

5.1.3.1 La mémoire partagée $MEM[1..(t+1)]$

La mémoire snapshot mem partagée par les processus simulés $p_1, \ldots, p_n$ est simulée par un objet snapshot MEM partagé par les simulateurs (donc MEM a $(t+1)$ entrées).

Plus spécifiquement, $MEM[i]$ est un registre qui contient un tableau avec une entrée par processus simulé p_j. Chaque $MEM[i][j]$ est constitué de deux champs : $MEM[i][j].value$ qui contient la dernière valeur de $mem[j]$ écrite par p_j et $MEM[i][j].sn$ qui contient le numéro de séquence correspondant. (Ce numéro de séquence, introduit par la simulation, est une valeur de contrôle qui sera utilisée pour produire une simulation cohérente des opérations $mem.\mathsf{snapshot}()$ invoquées par les processus simulés).

5.1.3.2 L'opération $\mathsf{sim\_write}_{i,j}()$

L'algorithme, noté $\mathsf{sim\_write}_{i,j}(v)$, exécuté par q_i pour simuler l'écriture par p_j de la valeur v dans $mem[j]$ est décrite dans la figure 5.3 [30]. Son code est relativement simple. Le simulateur q_i commence par incrémenter un numéro de séquence local $w\_sn_i[j]$ qui sera associé à la valeur v écrite par p_j dans $mem[j]$. Ensuite, q_i écrit la paire $(v, w\_sn_i[j])$ dans $mem_i[j]$ (où mem_i est sa copie locale de la mémoire partagée par les processus simulés) et termine en écrivant, de manière atomique, sa copie locale mem_i dans $MEM[i]$.

```
operation sim_write_{i,j}(v) :
(01) w_sn_i[j] ← w_sn_i[j] + 1 ;
(02) mem_i[j] ← (v, w_sn_i[j]) ;
(03) MEM[i] ← mem_i.
```

FIGURE 5.3 – $\mathsf{write}_{i,j}(v)$ exécutée par q_i pour simuler $\mathsf{write}(v)$ invoquée par p_j [30]

5.1.3.3 L'opération $\mathsf{sim\_snapshot}_{i,j}()$

Cette opération est implémentée par l'algorithme décrit dans la figure 5.4 [30].

Objets locaux et partagés additionnels Pour chaque processus simulé p_j, le simulateur q_i gère un générateur de numéros de séquence local $snap\_sn_i[j]$ utilisé pour associer un numéro de séquence à chaque $mem.\mathsf{snapshot}()$ qu'il simule pour p_j (ligne 04).

En plus de l'objet snapshot $MEM[1..(t+1)]$, les simulateurs $q_1, \ldots, q_{t+1}$ coopèrent à travers un tableau $SAFE\_AG[1..n, 0...]$ d'objets de type safe_agreement.

Principe de base de la simulation BG [27, 30] : obtenir une valeur cohérente Pour s'accorder sur la sortie de de la $snapsn$-ième invocation de $mem.\mathsf{snapshot}()$ effectuée par p_j, les simulateurs $q_1, \ldots, q_{t+1}$ utilisent l'objet $SAFE\_AG[j, snapsn]$.

Chaque simulateur q_i propose une valeur (notée $input_i$) à cet objet (ligne 05) et, grâce à sa propriété d'accord, cet objet renverra à tous les simulateurs la même sortie à la ligne 06. Pour assurer un progrès cohérent de la simulation, la valeur d'entrée $input_i$ proposée par le simulateur q_i à $SAFE\_AG[j, snapsn]$ est définie de la façon suivante.

- D'abord, q_i effectue un snapshot de MEM pour obtenir une vue cohérente de l'état de la simulation. La valeur de cette vue est stockée dans sm_i (ligne 01).
  Remarquons que $sm_i[x][y]$ est tel que (1) $sm_i[x][y].sn$ est le nombre d'écritures effectuées par p_y dans $mem[y]$ simulées jusque là par q_x et (2) $sm_i[x][y].value$ est la valeur de la dernière écriture dans $mem[y]$ par p_y telle que simulée par q_x.
- Ensuite, pour chaque p_y, q_i calcule $input_i[y]$. Pour cela, il extrait de $sm_i[1..t+1][y]$ la valeur écrite par le simulateur q_s le plus avancé dans sa simulation de p_y. Ceci est exprimé dans les lignes 02-03.

```
operation sim_snapshot_{i,j}() :
(01) sm_i ← MEM.snapshot() :
(02) for each y : 1 ≤ y ≤ n : do input_i[y] = sm_i[s][y].value
(03)          where ∀x : 1 ≤ x ≤ t + 1 : sm_i[s][y].sn ≥ sm_i[x][y].sn end for ;
(04) snap_sn_i[j] ← snap_sn_i[j] + 1 ; let snapsn = snap_sn_i[j] ;
(05) enter_mutex ; SAFE_AG[j, snapsn].propose_i(input_i) ; exit_mutex ;
(06) res ← SAFE_AG[j, snapsn].decide_i()
(07) return(res).
```

FIGURE 5.4 – $\mathsf{sim\_snapshot}_{i,j}()$ exécutée par q_i pour simuler $mem.\mathsf{snapshot}()$ invoquée par p_j [30]

Quand la valeur $input_i$ a été calculée, q_i la propose à $SAFE\_AG[j, snapsn]$ (ligne 05), puis renvoie la valeur décidée par cet objet (lignes 06-07).

La description précédente montre une caractéristique importante de la simulation BG. Une valeur $input_i[y] = sm_i[s][y].value$ proposée par un simulateur q_i peut être telle que $sm_i[s][y].sn > sm_i[i][y].sn$, c'est-à-dire que le simulateur q_s est plus avancé que q_i dans sa simulation de p_y. Cela ne pose pas de problème car, quand q_i simulera les opérations $mem.\mathsf{snapshot}()$ de p_y (s'il y en a) qui sont

entre les $(sm_i[i][y].sn)$-ième et $(sm_i[s][y].sn)$-ième opérations d'écriture de p_y, il obtiendra toujours une valeur qui a déjà été calculée et qui est stockée dans l'objet $SAFE\_AG[y, -]$ correspondant.

Principe de base de la simulation BG [27, 30] : de sans-attente à t-résilient Chaque simulateur q_i simule les n processus $p_1, \ldots, p_n$ "en parallèle" et de manière équitable, mais n'importe quel simulateur peut crasher. Le crash de q_i pendant qu'il est engagé dans la simulation de mem.snapshot() pour plusieurs processus p_j, $p_{j'}$, etc., peut provoquer leur blocage définitif, c'est-à-dire leur crash. Ce blocage est dû au fait que chaque objet $SAFE\_AG[j, -]$ garantit la terminaison des invocations de $SAFE\_AG[j, -]$.decide() uniquement si aucun simulateur ne crashe durant l'exécution de l'opération $SAFE\_AG[j, -]$.propose() (ligne 05 de la figure 5.4).

L'idée de la simulation BG pour résoudre ce problème consiste à obliger un simulateur à ne pas être engagé dans plus d'une invocation $SAFE\_AG[-, -]$.propose() à la fois. Donc si q_i crashe durant son exécution de $SAFE\_AG[j, -]$.propose(), il ne peut provoquer que le crash de p_j. Ceci est obtenu en utilisant un objet d'exclusion mutuel additionnel offrant les opérations enter_mutex et exit_mutex. (Remarquons que cet objet est purement local à chaque simulateur : il résout les conflits entre les différents threads de chaque simulateur et ne concerne en rien la mémoire partagée par les différents simulateurs.)

De la t-résilience à la terminaison sans-attente Comme exemple, considérons que nous avons un algorithme t-résilient qui résout le t-accord ensembliste sur n processus. Nous obtenons alors un algorithme sans-attente qui résout le t-accord ensembliste pour $t+1$ processus de la manière suivante. Chaque simulateur q_i ($1 \leq i \leq t+1$) reçoit initialement une valeur proposée v_i et les objets de base $SAFE\_AG[1..n, 0]$ sont utilisés par les $t + 1$ simulateurs pour déterminer la valeur proposée par chaque p_j de la manière suivante. Pour chaque j, $1 \leq j \leq n$, le simulateur q_i invoque d'abord $SAFE\_AG[j, 0]$.propose$_i(v_i)$, puis $SAFE\_AG[j, 0]$.decide$_i()$ qui lui renvoie une valeur qu'il considère comme la valeur proposée par p_j. Pour chaque j, tous les simulateurs obtiennent donc la même valeur pour p_j. De plus, cette valeur est une des $t + 1$ valeurs proposées par les simulateurs. Finalement, le simulateur q_i peut décider une valeur quelconque parmi celles décidées par les processus p_j qu'il simule. (On peut remarquer facilement pourquoi la simulation BG ne marche pas pour les tâches colorées.) Une démonstration formelle de cette réduction (basée sur des automates input/output) est fournie dans [30].

De la terminaison sans-attente à la t-résilience Pour les tâches non colorées, la t-résilience peut être réduite facilement à la terminaison sans-attente de la manière suivante. Un sous-ensemble de $t + 1$ processus du système exécute alors l'algorithme sans-attente et stocke sa sortie dans un registre partagé. Comme il ne peut pas y avoir plus de t crashes, au moins l'un d'entre eux terminera son exécution. Chaque processus peut ensuite décider n'importe quelle valeur décidée par un des processus ayant exécuté l'algorithme sans-attente.

5.1.4 La simulation BG étendue

Cette section étend les algorithmes précédents pour fournir une implémentation de la simulation BG étendue. Notre but est d'obtenir une simulation aussi simple que possible. Pour cela, nous procédons de manière incrémentale en enrichissant la simulation BG classique. L'implémentation proposée utilise le même objet snapshot MEM et la même opération sim_write$_{i,j}()$ (figure 5.3) que la simulation BG classique. Elle utilise aussi le même tableau $SAFE\_AG[1..n, 0...]$ constitué d'objets safe_agreement.

Elle présente les objets partagés additionnels utilisés, les principes sur lesquels se base la simulation de mem.snaspshot() par un simulateur q_i pour un processus simulé p_j ainsi que l'algorithme (noté e_sim_snapshot$_{i,j}()$) qui l'implémente.

5.1.4.1 Les objets partagés additionnels

En plus de MEM et $SAFE\_AG[1..n, 0...]$, la mémoire partagée par les simulateurs $q_1, \ldots, q_{t+1}$ contient les objets suivants.

- $ARBITER[1..t+1, 0...]$ est un tableau d'objets de type arbiter. Le propriétaire des objets $ARBITER[j, -]$ est le simulateur q_j ($1 \leq j \leq t+1$).
  L'objet $ARBITER[j, snapsn]$ est utilisé par un simulateur q_i quand il simule son $snapsn$-ième invocation mem.snapshot() pour le processus simulé p_j pour $1 \leq j \leq t+1$. (Comme nous le verrons, quand $t+1 < j \leq n$, la simulation de mem.snapshot() pour p_j ne nécessite pas l'aide d'un objet arbiter.)
- $ARB\_VAL[1..t+1, 0...][0..1]$ est un tableau de registres atomiques. La paire de registres atomiques $ARB\_VAL[j, snapsn][0..1]$ est associée à l'objet $ARBITER[j, snapsn]$.
  Le but de $ARB\_VAL[j, snapsn][1]$ est de contenir la valeur qui doit être renvoyée pour la $snapsn$-ième invocation mem.snapshot() par le processus simulé p_j si le propriétaire q_j est désigné comme gagnant par l'objet $ARBITER[j, snapsn]$ correspondant. Si le propriétaire q_j n'est pas le gagnant, la valeur qui doit être renvoyée est celle stockée dans $ARB\_VAL[[j, snapsn][0]$.

5.1.4.2 L'opération e_sim_snapshot$_{i,j}$()

L'algorithme enrichi Le code de l'algorithme qui implémente l'opération e_sim_snapshot$_{i,j}$() exécutée par q_i pour simuler une opération mem.snapshot() invoquée par p_j est décrite dans la figure 5.5. Ses quatre premières lignes et sa dernière ligne sont exactement les mêmes que celles de la figure 5.4. Les lignes 05-06 sont remplacées par les nouvelles lignes N01-N11 qui constituent l'"addition" qui permet de passer de la simulation BG classique à la simulation BG étendue.

Principe de base Chaque processus p_j ($1 \leq j \leq n$) est simulé par chaque simulateur q_i ($1 \leq i \leq t+1$) comme dans la simulation BG classique, mais chaque processus simulé p_j tel que $1 \leq j \leq t+1$ est associé à exactement un simulateur qui est son "propriétaire" : q_i est le propriétaire de p_j si $j = i$ (et donc aussi le propriétaire des objets $ARBITER[j, -]$ correspondants). Le but est, pour chaque $snapsn \geq 0$, d'associer une seule valeur renvoyée pour les $snapsn$-ièmes invocations de e_sim_snapshot$_{i,j}$() par les simulateurs. L'idée est d'utiliser la notion de propriétaire pour "court-circuiter" l'utilisation de l'objet $SAFE\_AG[j, snapsn]$ dans des circonstances appropriées.

L'opération e_sim_snapshot$_{i,j}$() pour les processus simulés p_j tels que $t+2 \leq j \leq n$ est exactement la même que sim_snapshot$_{i,j}$(). Cela apparait dans les lignes N01-N02 qui sont les mêmes que les lignes 06-07 de la figure 5.4 (dans ce cas, il n'y a pas de notion de propriétaire).

Les nouvelles lignes N03-N10 traitent le cas des processus simulés qui ont un propriétaire, c'est-à-dire les processus $p_1, \ldots, p_{t+1}$. L'idée est la suivante : si q_i ne crashe pas, p_i ne doit pas crasher. De cette manière, si q_i est correct, p_i terminera toujours quel que soit le comportement des autres simulateurs. Pour cela, q_i d'un coté et tous les autres simulateurs de l'autre sont en compétition pour définir la valeur du snapshot renvoyée par les $snapsn$-ièmes invocations e_sim_snapshot$_{i,j}$() effectuées par chacun d'entre eux. Pour atteindre ce but, les objets $ARBITER[j, snapsn]$ et $ARB\_VAL[j, snapsn]$ sont utilisés de la manière suivante.

Tous les simulateurs invoquent $ARBITER[j, snapsn]$.arbitrate$_{i,j}$() (à la ligne N04 si q_i est le propriétaire, à la ligne N09 s'il ne l'est pas). Selon les spécifications du type arbiter, ces invocations ne doivent pas renvoyer des valeurs différentes et terminent au moins quand q_j est correct et invoque cette opération (comme indiqué dans la spécification, il y a d'autres cas où les invocations doivent terminer). Finalement, la valeur renvoyée indique si le propriétaire est le gagnant (1) ou non (0).

Si le gagnant est le propriétaire q_j, la valeur renvoyée par les $snapsn$-ièmes invocations de l'opération e_sim_snapshot$_{i,j}()$ (une invocation par simulateur) est la valeur $input_j$ calculée par le propriétaire. Cette valeur est stockée dans le registre atomique $ARB\_VAL[j, snapsn][1]$ (ligne N03).

Si le propriétaire n'est pas le gagnant, la valeur renvoyée est celle déterminée par les autres simulateurs qui ont invoqué $SAFE\_AG[j, snapsn]$.propose$_i(input_i)$ et $SAFE\_AG[j, snapsn]$.decide$_i()$ (lignes N07 et N08). La valeur qu'ils ont calculé a été déposée dans $ARB\_VAL[j, snapsn][0]$ (ligne N08).

Il est important de remarquer que q_j n'invoque pas propose$_j()$ et decide$_j()$ pour le processus dont il est propriétaire. De plus, le simulateur q_j est le seul qui peut écrire dans $ARB\_VAL[j, snapsn][1]$ tandis que les autres simulateurs ne peuvent écrire que dans $ARB\_VAL[j, snapsn][0]$.

```
operation e_sim_snapshot_{i,j}() :
(01)   sm_i ← MEM.snapshot() :
(02)   for each y : 1 ≤ y ≤ n : do input_i[y] = sm_i[s, y].value
(03)                                  where ∀x : 1 ≤ x ≤ t + 1 : sm_i[s, y].sn ≥ sm_i[x, y].sn end for ;
(04)   snap_sn_i[j] ← snap_sn_i[j] + 1 ; let snapsn = snap_sn_i[j] ;
(N01)  if (j > t + 1) then enter_mutex ; SAFE_AG[j, snapsn].propose_i(input_i) ; exit_mutex ;
(N02)                      res ← SAFE_AG[j, snapsn].decide_i()
(N03)  elseif (i = j) then ARB_VAL[j, snapsn][1] ← input_i ;
(N04)                      enter_mutex ; win ← ARBITER[j, snapsn].arbitrate_{i,j}() ; exit_mutex ;
(N05)                      if (win = 1) then res ← input_i
(N06)                                   else res ← ARB_VAL[j, snapsn][0] end if ;
(N07)                 else enter_mutex ; SAFE_AG[j, snapsn].propose(input_i) ; exit_mutex ;
(N08)                      ARB_VAL[j, snapsn][0] ← SAFE_AG[j, snapsn].decide_i() ;
(N09)                      r ← ARBITER[j, snapsn].arbitrate_{i,j}() ;
(N10)                      res ← ARB_VAL[j, snapsn][r]
(N11)  end if ;
(07)   return(res).
```

FIGURE 5.5 – L'opération e_sim_snapshot$_{i,j}()$ exécutée par q_i pour simuler mem.snapshot() invoquée par p_j

Pour résumer, si un simulateur q_i crashe, il entraine le crash d'au plus un processus simulé grâce à l'utilisation de l'exclusion mutuelle sur les invocations de propose() des objets safe_agreement. Si le simulateur q_i crashe, $1 \leq i \leq t+1$, du point de vue des processus simulés, il peut n'entrainer (1) aucun crash (si il crashe en dehors d'une section critique), (2) le crash de p_i (si il crashe durant l'exécution de arbitrate$_{i,j}()$ dans la section critique de la ligne N04), (3) le crash d'un processus p_j tel que $1 \leq j \neq i \leq t+1$ (ceci ne peut arriver que si q_j a déjà crashé sans entrainer le blocage de p_j, et q_i crashe dans la section critique de la ligne N08), ou (4) le crash d'un des processus p_{t+2}, ..., p_n (si il crashe à la ligne N01 dans la section critique).

t-Résilience vs terminaison sans-attente Étant donné un algorithme de simulation BG dans lequel un processus simulé p_j ($1 \leq j \leq t+1 \leq n$) ne peut être bloqué indéfiniment que si le simulateur q_j crashe et une tâche t-résiliente, Gafni montre dans [55] la tâche sans-attente équivalente sur $t+1$ processus.

5.1.5 Démonstration de l'algorithme de simulation BG étendue

Lemme 5.8. *Un simulateur ne peut pas bloquer la progression de plus d'un processus simulé à un instant donné.*

Démonstration. Un simulateur ne peut bloquer la simulation d'un processus que durant une opération e_sim_snapshot(), quand le simulateur utilise un safe_agreement (lignes N01-N02 ou N07-N08) ou un objet arbiter parce que c'est son propriétaire (ligne N04). Toutes ces invocations sont placées en exclusion

mutuelle. Un simulateur ne peut donc pas bloquer la progression de plus d'un processus simulé à un instant donné. □

Lemme 5.9. *Le processus simulé p_i n'est jamais bloqué au simulateur q_i.*

Démonstration. L'opération e_sim_snapshot(), quand elle est invoquée par le simulateur q_i pour le processus simulé p_i (ligne N03, $i = j$) n'inclut aucune instruction d'attente et n'utilise pas d'objet safe_agreement. À cause des propriétés du type arbiter, il ne peut pas être bloqué durant son invocation de arbitrate(). Le processus simulé p_i ne peut donc jamais être bloqué au simulateur q_i. □

Lemme 5.10. *Chaque simulateur reçoit les valeurs de décision d'au moins $n - t$ processus simulés.*

Démonstration. Parce qu'au plus t simulateurs peuvent crasher et qu'un simulateur ne peut pas bloquer plus d'un processus simulé à la fois (lemma 5.8), chaque simulateur peut exécuter le code d'au moins $n - t$ processus simulés sans être bloqué indéfiniment. Parce que l'algorithme simulé est t-résilient, ces $n - t$ processus décideront alors une valeur. □

Lemme 5.11. *Tous les simulateurs qui renvoient une valeur pour le k-ième snapshot du processus simulé p_j renvoient la même valeur.*

Démonstration. Si le processus simulé p_j n'a aucun propriétaire ($j > t + 1$), alors les propriétés des objets safe_agreement garantissent que la même valeur de snapshot est toujours renvoyée (lignes N01-N02 de la figure 5.5).

Si le propriétaire du processus simulé p_j choisit la valeur qu'il a calculé pour le k-ième snapshot de p_j, il a écrit cette valeur dans $ARB\_VAL[j, snapsn][1]$ (ligne N03) et est le gagnant de l'objet arbiter (ligne N05). Tous les autres simulateurs liront alors cette valeur (ligne N10).

Si le processus simulé a un autre propriétaire mais un autre simulateur choisit la valeur qu'il a calculé pour le k-ième snapshot de p_j, ce simulateur s'est déjà mis d'accord sur une valeur avec tous les autres non-propriétaires (objet safe_agreement, lignes N07-N08) et il est gagnant à l'objet arbiter (lignes N09-N10). Tous les non-propriétaires écriront alors la même valeur dans $ARB\_VAL[j, snapsn][0]$ (ligne N08) et le propriétaire la lira (ligne N06).

Tous les simulateurs qui renvoient une valeur pour le k-ième snapshot du processus simulé p_j renvoient donc la même valeur. □

Lemme 5.12. *Un processus simulé ne peut pas décider plus d'une valeur sur n'importe quel simulateur.*

Démonstration. Parce que chaque simulateur calcule la même valeur pour chaque snapshot et parce que les snapshots sont les seules parties non-déterministes des codes des processus simulés, tous les simulateurs qui décident une valeur pour un processus simulé donné décident la même valeur. □

Lemme 5.13. *Les séquences de toutes les écritures et tous les snapshots pour chaque processus simulé correspondent à une exécution correcte de l'algorithme simulé.*

Démonstration. Chaque simulateur qui n'est pas bloqué durant la simulation d'un processus le simule de la même manière (mêmes valeurs écrites et mêmes snapshots lus, lemme 5.11).

Quand le simulateur q_i exécute e_sim_snapshot() pour p_j (c'est-à-dire la simulation d'un snapshot pour p_j), il stocke dans sa variable $input_i$ les valeurs écrites par les simulateurs qui ont le plus avancé pour chaque processus simulé(figure 5.3 et lignes 01-03 de la figure 5.5). Il ne peut choisir sa propre valeur de snapshot $input_i$ que si aucun autre processus n'a déjà terminé son exécution de ce e_sim_snapshot() (le lemme 5.11 implique que les objets safe_agreement ont un effet "mémoire"). Pour chaque e_sim_snapshot(), q_i renvoie donc une valeur de $input$ calculée par lui-même ou un autre simulateur. Remarquons que, à l'instant auquel cette valeur de $input$ a été déterminée, aucun simulateur n'avait terminé son e_sim_snapshot() correspondant. (Si ce n'était pas le cas, ce simulateur aurait fourni

sa propre valeur de $input$ aux autres simulateurs.) Parce que les processus sont simulés de manière déterministe, la valeur de $input$ qui est renvoyée contient la dernière valeur écrite par p_j du point de vue de q_i. Cela montre que l'ordre de chaque processus simulé est respecté.

Pour garantir que la simulation est correcte, nous devons maintenant montrer que les écritures et les snapshots de tous les processus simulés peuvent être linéarisés. Le point de linéarisation des écritures est placé à la ligne 03 de la figure 5.3 du premier simulateur qui l'exécute. Le point de linéarisation des snapshots est placé à la ligne 01 de la figure 5.5 du simulateur q_i qui impose sa valeur de $input_i$.

Parce que le simulateur q_i qui impose sa valeur de $input_i$ dans une opération e_sim_snapshot() lit les valeurs les plus avancées au moment du snapshot (lignes 02-03 de la figure 5.5) et parce qu'une fois qu'un simulateur termine l'exécution de e_sim_snapshot(), la valeur pour ce e_sim_snapshot() ne peut plus changer (lemme 5.11), la linéarisation correspond à une linéarisation d'une exécution correcte de l'algorithme simulé. □

Théorème 5.14. *Les algorithmes de simulation BG étendue décrits dans les figures 5.3 et 5.5 sont corrects.*

Démonstration. Les lemmes 5.9, 5.10, 5.12 et 5.13 montrent que les algorithmes de simulation BG étendue présentés ici sont corrects. □

5.2 Puissance multiplicative du x-consensus

Comme indiqué précédemment, les simulations BG classique et étendue ne traitent que le cas des systèmes dans lesquels les processus n'ont accès qu'à des registres (sous la forme d'objets snapshot). Nous nous intéressons maintenant au cas où les processus ont accès à des objets plus puissants que des registres.

Paramètres du modèle Notons $ASM(n, t, x)$ un modèle de système en mémoire partagée constitué de n processus, parmi lesquels jusqu'à t peuvent crasher ($1 \leq t < n$) et où les processus communiquent et coopèrent en utilisant des registres atomiques et des objets x-consensus, avec $1 \leq x \leq n$ (le consensus peut être résolu dans n'importe quel sous-ensemble de x processus) [2]. Le paramètre t mesure la puissance de l'*adversaire* (qui peut crasher arbitrairement jusqu'à t processus), tandis que, du coté de "l'aide", x mesure la *puissance de synchronisation sans-attente* dont les processus peuvent bénéficier de la part des objets partagés. Cette section considère la puissance d'un système, du point de vue de la calculabilité, des modèles de système $ASM(n, t, x)$ tels que $n > t$ et $n \geq x$ par rapport aux tâches non colorées.

En utilisant cette notation, la simulation BG montre que $ASM(n, t, 1)$ et $ASM(t + 1, t, 1)$ ont la même puissance pour les tâches de décision. En fait, quand on considère des systèmes dans lesquels les processus n'ont accès qu'à des registres ($x = 1$) et dans lesquels le nombre de crashes possibles est borné (t), la simulation BG montre que le paramètre crucial du système n'est pas le nombre n de processus, mais t. Quel que soit le nombre de processus, la t-résilience peut être caractérisée par la terminaison sans-attente sur $t + 1$ processus [27]. La simulation BG traite donc une un aspect du problème (elle considère $x = 1$ et traite l'équivalence entre la paire (n, t) et la paire $(t + 1, t)$), tandis que cette section traite l'autre face du problème (elle considère un nombre fixe n de processus et traite l'équivalence entre la paire (t', x) et la paire $(t, 1)$ en comparant $ASM(n, t', x)$ et $ASM(n, t, 1)$).

Cette section se concentre donc sur les paramètres t et x du modèle. Soient $ASM(n, t_1, x_1)$ et $ASM(n, t_2, x_2)$ deux modèles de systèmes tels que $n > \max(t_1, t_2)$. La question principale traitée dans cette section est la suivante : "Quelles conditions doivent respecter les paramètres t_1, x_1, t_2 et x_2 pour que toute tâche (non colorée) qui peut être résolue (respectivement qui est impossible à résoudre) dans

2. Quand $x > t$, le consensus peut alors être résolu sur l'ensemble des processus du système.

$ASM(n, t_1, x_1)$, puisse être résolue (respectivement soit impossible à résoudre) dans $ASM(n, t_2, x_2)$?" Nous répondons à cette question en apportant les deux contributions suivantes.

Contribution #1 La première contribution est une condition nécessaire et suffisante qui répond à la question précédente. Cette condition est $\lfloor \frac{t_1}{x_1} \rfloor \geq \lfloor \frac{t_2}{x_2} \rfloor$. Ce résultat a plusieurs conséquences : $ASM(n, t_1, x_1)$ et $ASM(n, t_2, x_2)$ sont équivalents (ont la même puissance de calculabilité) par rapport aux tâches non colorées si et seulement si $\lfloor \frac{t_1}{x_1} \rfloor = \lfloor \frac{t_2}{x_2} \rfloor$.

Pour cela, nous utilisons deux algorithmes de simulation. Le premier est une simulation du modèle $ASM(n, t', x)$ dans $ASM(n, t, 1)$. Cette simulation nécessite que $t \leq \lfloor \frac{t'}{x} \rfloor$. Le deuxième est une simulation de $ASM(n, t, 1)$ dans $ASM(n, t', x)$. Cette simulation nécessite que $t \geq \lfloor \frac{t'}{x} \rfloor$. Cela montre que $ASM(n, t', x)$ et $ASM(n, t, 1)$ sont équivalents si et seulement si $(t \times x) \leq t' \leq (t \times x) + (x - 1)$.

Pour illustrer l'intérêt de ces équivalences, considérons les exemples suivants qui sont des conséquences immédiates de nos résultats.

- Considérons le modèle de système $ASM(n, n-1, n-1)$, c'est-à-dire un système dans lequel les processus ont accès à des objets $n-1$-consensus (le consensus ne peut donc pas être résolu dans ce système de n processus). De plus, comme $t = n - 1$, un algorithme qui résout une tâche dans ce système doit être sans-attente.
  Cette section montre que, pour toute tâche non colorée T, les résultats de possibilité et d'impossibilité sont les mêmes dans $ASM(n, n-1, n-1)$ et dans $ASM(n, 1, 1)$, et plus généralement dans tout modèle $ASM(n, t, t)$. Comme, pour tout n, le consensus ne peut pas être résolu dans $ASM(n, n-1, n-1)$ (c'est-à-dire ne peut pas être résolu de manière sans-attente avec des objets $(n-1)$-consensus), il ne peut pas non plus être résolu dans $ASM(n, 1, 1)$ (c'est-à-dire de manière 1-résiliente dans le modèle doté uniquement de registres). Cela constitue une nouvelle démonstration que, quelles que soient les valeurs de n et de t, (a) le consensus ne peut pas être résolu de manière t-résiliente dans $ASM(n, t, t)$ et, $\forall\ t' < t$,(b) le modèle $ASM(n, t', t)$ et le modèle sans pannes et doté de registres $ASM(n, 0, 1)$ sont équivalents.
  Plus généralement, ces simulations nous fournissent une démonstration générale de l'impossibilité de tâches t-résilientes quand les processus peuvent communiquer en utilisant des registres et des objets x-consensus. Une tâche qui ne peut pas être résolue dans $ASM(n, t, 1)$ ne peut pas non plus être résolue dans $ASM(n, t', x)$ pour les paires (t', x) telles que $t \leq \lfloor \frac{t'}{x} \rfloor$, et une tâche qui peut être résolue dans $ASM(n, t, 1)$ peut aussi être résolue dans $ASM(n, t', x)$ pour les paires (t', x) telles que $t \geq \lfloor \frac{t'}{x} \rfloor$.
- Soit T_k une tâche dont le nombre d'accord ensembliste est k. Cette tâche peut être résolue dans $ASM(n, t, 1)$ pour $t < k$ (car le k-accord ensembliste peut être résolu dans $ASM(n, t, 1)$ avec $t < k$), et ne peut pas être résolue dans $ASM(n, t, 1)$ pour $t \geq k$ (car le k-accord ensembliste ne peut pas être résolu dans $ASM(n, t, 1)$ avec $t \geq k$ [27, 75, 112]). Nos résultats montrent que T_k peut être résolu dans tout système $ASM(n, t', x)$ tel que $\lfloor \frac{t'}{x} \rfloor \leq (k-1)$, c'est-à-dire $t' \leq (k-1) \times x + (x-1) = k \times x - 1$ si x est fixé ou $x \geq \frac{t'+1}{k}$ si t' est fixé.

Contribution #2 La deuxième contribution utilise résultats précédents : c'est une généralisation de la simulation BG. Nous montrons que toute tâche qui peut être résolue (respectivement qui est impossible) dans $ASM(n, t, x)$ peut être résolue (respectivement est impossible) dans $ASM(t+1, t, x)$. (Le cas $x = 1$ correspond à la simulation BG.)

Une caractéristique notable de ces résultats est qu'ils sont tous obtenus par réduction algorithmique. Cette section, inspirée par la simulation BG, la renforce donc en introduisant une autre simulation puissante pour obtenir des résultats de possibilité et d'impossibilité pour les tâches de décision dans les systèmes asynchrones soumis aux pannes. Elle généralise, unifie et étend également des résultats

précédents. Comme noté précédemment, la simulation BG et la simulation proposée ici sont deux aspects d'un même problème.

5.2.1 Modèle de système : $ASM(n, t, x)$

Le modèle en mémoire partagée asynchrone $ASM(n, t, x)$ ($1 \leq t < n$) a été décrit brièvement dans l'introduction de la section. Quand $x = 1$, ce modèle est le même que celui de la section précédente : il est constitué de n processus asynchrones p_1, ..., p_n qui communiquent en utilisant des registres partagés [3]. Au plus t parmi ces n processus peuvent crasher (ces processus peuvent être choisis arbitrairement par "l'adversaire" du système).

Quand $x > 1$, les processus peuvent en plus accéder à autant d'objets x-consensus que nécessaire. Un objet x-consensus permet de résoudre le problème du consensus entre tous les processus qui l'accèdent, mais il ne peut pas être accédé par plus de x processus. Ce sous-ensemble de x processus est défini statiquement pour chaque objet x-consensus. En utilisant une notation semblable aux tableaux, un tel objet est noté $x\_cons[a]$. Un processus p_i, qui peut accéder à l'objet $x\_cons[a]$, y accède en invoquant $x\_cons[a].\mathsf{x\_cons\_propose}(v)$ où v est la valeur qu'il propose à cet objet consensus. Si p_i est correct, cette invocation renvoie la valeur décidée par $x\_cons[a]$.

Rôle d'un simulateur Comme dans la section précédente, chaque simulateur q_i reçoit le code de tous les processus simulés $p_1, \ldots, p_n$. Il gère n threads, chacun associé à un processus simulé, et exécute localement ces threads d'une manière équitable. Il gère aussi une copie locale mem_i de la mémoire snapshot mem partagée par les processus simulés. Contrairement aux systèmes considérés dans la section précédente, si le système simulé est $ASM(n, t, x)$ avec $x > 1$, chaque simulateur garde aussi une copie locale $x\_cons[a]_i$ de chaque objet x-consensus partagé par ces processus.

Le code d'un processus simulé p_j contient des invocations d'écriture dans $mem[j]$ et des invocations de $mem.\mathsf{snapshot}()$. Si le système simulé est $ASM(n, t, x)$ avec $x > 1$, il contient également des invocations de $x\_cons[a].\mathsf{x\_cons\_propose}()$ (si p_j est un des x processus qui peuvent accéder à $x\_cons[a]$). Ce sont les seules opérations utilisées par les processus simulés $p_1, \ldots, p_n$ pour coopérer. Le cœur de la simulation est donc la définition d'algorithmes pour simuler ces opérations correctement (dans $ASM(n, t', x')$) quand elles sont invoquées par un processus p_j (dans $ASM(n, t, x)$). Ces algorithmes de simulation sont notés $\mathsf{sim\_write}_{i,j}()$, $\mathsf{sim\_snapshot}_{i,j}()$ et $\mathsf{sim\_x\_cons\_propose}_{i,j}()$.

5.2.2 Simulation de $ASM(n, t', x)$ dans $ASM(n, t, 1)$

Soit A un algorithme qui résout une tâche T dans $ASM(n, t', x)$. Cela signifie que A est un algorithme t'-résilient qui qui peut utiliser des objets x-consensus. Cette section présente un algorithme qui, étant donné A en entrée, produit un algorithme t-résilient A' qui résout T dans $ASM(n, t, 1)$. En supposant $\max(t, t') < n$, cette simulation nécessite $t \leq \lfloor \frac{t'}{x} \rfloor$.

La simulation BG (présentée dans la section précédente) nous fournit des implémentations appropriées des opérations $mem[j].\mathsf{write}()$ et $mem.\mathsf{snapshot}()$ quand nous voulons simuler, dans $ASM(t+1, t, 1)$, un algorithme qui résout une tâche non colorée dans $ASM(n, t, 1)$. Comme nous allons le voir, l'algorithme proposé, qui simule dans $ASM(n, t, 1)$ un algorithme A conçu pour $ASM(n, t', x)$, est une extension simple de la simulation BG. Il emprunte à la simulation BG les opérations $mem[j].\mathsf{write}()$ et $mem.\mathsf{snapshot}()$ et fournit en plus l'implémentation de l'opération $\mathsf{x\_cons\_propose}()$ d'un objet x-consensus. Les opérations $mem[j].\mathsf{write}()$ et $mem.\mathsf{snapshot}()$ ayant déjà été présentées dans la section précédente, nous présenterons ici uniquement l'implémentation de l'opération $\mathsf{x\_cons\_propose}()$.

3. Rappelons que les objets snapshots peuvent être implémentés de manière sans-attente en utilisant des registres partagés [1, 23].

5.2.2.1 Simulation de $x\_cons[a]$.x_cons_propose()

Comme indiqué précédemment, les processus simulés p_j coopèrent en utilisant écrivant et lisant une mémoire snapshot mem et en accédant des objets x-consensus. Soit $x\_cons[a]$ un tel objet. Rappelons qu'un tel objet est un objet de décision (un processus invoque $x\_cons[a]$.x_cons_propose(v) au plus une fois).

L'implémentation de $x\_cons[a]$.x_cons_propose(v) (faite par un des x processus p_j simulés qui peuvent accéder à $x\_cons[a]$) par un simulateur q_i est décrite dans dans la figure 5.6. La valeur décidée pour $x\_cons[a]$ est calculée par les simulateurs à l'aide d'un objet safe_agreement noté $XSAFE\_AG[a]$ (le type d'objet safe_agreement et son implémentation sont décrits au début de ce chapitre). Cette valeur est ensuite stockée par un simulateur q_i dans une variable locale $xres_i[a]$ (initialisée à $\perp$).

Plus précisément, si q_i connait déjà cette valeur quand il invoque $x\_cons[a]$.x_cons_propose(v) pour le processus simulé p_j (c'est le cas quand q_i a déjà invoqué $x\_cons[a]$.x_cons_propose() pour un autre processus simulé $p_{j'}$), q_i la renvoie directement. Sinon, q_i propose v à l'objet safe_agreement $XSAFE_AG[a]$puis renvoie la valeur décidée dans cet objet. De plus, comme $XSAFE\_AG[a]$ n'est utilisable qu'une seule fois par chaque simulateur q_i, il doit invoquer au plus une fois les opérations $XSAFE\_AG[a]$.sa_propose$_i$() et $XSAFE\_AG[a]$.sa_decide$_i$() (dans cet ordre). Pour empêcher q_i d'invoquer ces opérations plusieurs fois, l'accès à la variable locale $xres_i[a]$ est protégé par un mécanisme d'exclusion mutuelle (local à q_i et implémenté par enter_mutex2 et exit_mutex2).

```
operation sim_x_cons_propose^a_{i,j}(v) :
(01) enter_mutex2 ;
(02) if (xres_i[a] = ⊥) then enter_mutex1 ; XSAFE_AG[a].sa_propose_i(v) ; exit_mutex1 ;
(03)                          xres_i[a] ← XSAFE_AG[a].sa_decide_i()
(04) end if ;
(05) exit_mutex2 ;
(06) return(xres_i[a]).
```

FIGURE 5.6 – sim_x_cons_propose$^a_{i,j}$() exécutée par q_i pour simuler $x\_cons[a]$.x_cons_propose() invoquée par p_j

Si le simulateur q_i crashe durant l'exécution de $XSAFE\_AG[a]$.sa_propose(v), il peut bloquer indéfiniment la simulation des x processus qui accèdent à l'objet simulé $x\_cons[a]$. L'exclusion mutuelle (locale à q_i) réalisée par enter_mutex2 and exit_mutex2 garantit qu'un simulateur peut simuler au plus une invocation $x\_cons[]$.x_cons_propose() à la fois. Concernant les invocations de x_cons_propose(), si τ simulateurs crashent, ils peuvent donc entrainer le crash d'au plus $\tau \times x$ processus simulés.

D'un autre coté, le crash de q_i pendant les exécutions simultanées de $XSAFE\_AG[a]$.sa_propose$_i$() (pour un processus p_j) et de $SAFE\_AG[j', snapsn]$.sa_propose$_i$() (pour un autre processus $p_{j'}$) pourrait entrainer le crash (le blocage de la simulation) de $x+1$ processus simulés. Pour éviter cela, un simulateur q_i n'est pas autorisé à exécuter plus d'une opération sa_propose$_i$() à la fois. Ceci est réalisé en appelant enter_mutex1 et exit_mutex1 avant et après $XSAFE\_AG[a]$.sa_propose$_i$() (comme dans l'opération e_sim_snapshot$_{i,j}$() exécutée par q_i pour simuler mem.snapshot() invoquée par p_j).

Le crash d'un simulateur q_i peut donc entrainer (1) le crash d'au plus un processus simulé si q_i crashe durant l'exécution de $SAFE\_AG[j, snapsn]$.sa_propose$_i$() durant la simulation d'un snapshot, ou (2) le crash d'au plus x processus simulés si q_i crashe durant l'exécution de $XSAFE\_AG[a]$.sa_propose$_i$()) durant la simulation d'une opération x_cons_propose(). Comme au plus t simulateurs peuvent crasher, c'est la raison pour laquelle la simulation nécessite $t \leq \lfloor \frac{t'}{x} \rfloor$.

5.2.2.2 Démonstration de la simulation de $ASM(n, t', x)$ dans $ASM(n, t, 1)$

Ce qui doit être démontré : coté vivacité Pour garantir la terminaison d'un algorithme t'-résilient, il faut qu'il n'y ait pas plus de t' crashes (parmi les processus simulés). Nous montrons ici que, quand

au plus $t \leq \lfloor \frac{t'}{x} \rfloor$ simulateurs peuvent crasher (c'est-à-dire $t' \geq t \times x$), tout simulateur correct q_i peut exécuter, sans rester bloquer indéfiniment durant cette exécution, les codes d'au moins $(n - t')$ processus simulés q_j qui accèdent à des objets snapshot et à des objets x-consensus.

Ce qui doit être démontré : coté sûreté Pour la sûreté, nous avons :

- Il doit être démontré qu'un processus simulé p_j qui exécute mem.snapshot() obtient la même valeur à chaque simulateur q_i. De plus, les valeurs de snapshot renvoyées doivent être cohérentes avec les opérations d'écriture $mem[j']$.write() invoquées par les processus simulés $p_{j'}$.
- En plus des objets snapshot, les processus simulés p_j peuvent accéder des objets x-consensus $xcons[a]$ (opération x_cons_propose()). Il doit être démontré que, pour chaque objet $xcons[a]$, les x processus simulés qui peuvent y accéder reçoivent la même valeur.

Lemme 5.15. *Le crash d'un simulateur q_i ne peut pas entrainer le blocage permanent de plus de x processus simulés p_j.*

Démonstration. Observons que le crash d'un simulateur q_i ne peut bloquer la simulation d'un processus p_j que quand q_i accède à un objet de type safe_agreement. Cela ne peut arriver que quand q_i exécute une opération sim_snapshot() (pour simuler mem.snapshot() invoquée par p_j), ou une opération sim_x_cons_propose() (pour simuler une invocation de x_cons_propose() sur un objet $xcons[a]$).

Si q_i crashe durant l'exécution de sim_snapshot(), à cause du mécanisme d'exclusion mutuelle (mutex1), il ne peut bloquer définitivement que p_j. Si q_i crashe durant l'exécution de sim_x_cons_propose(), à cause du mécanisme d'exclusion mutuelle (mutex2), il ne peut bloquer définitivement que les x processus qui accèdent à $xcons[a]$. De plus, comme (grâce à mutex1) q_i est empêché d'accéder à plus d'un objet de type safe_agreement à la fois, son crash ne peut pas entrainer le blocage de plus de x processus simulés. □

Rappelons que les processus simulés $p_1, \ldots, p_n$ doivent résoudre une tâche de décision. Cela signifie que, dans $ASM(n, t', x)$, chaque p_j qui ne crashe pas doit *décider* une valeur.

Lemme 5.16. *Soit $t \leq \lfloor \frac{t'}{x} \rfloor$. Chaque simulateur correct q_i calcule la valeur de décision d'au moins $(n - t')$ processus simulés p_j.*

Démonstration. Parce que (a) au plus t simulateurs peuvent crasher, (b) un simulateur peut bloquer au plus x processus simulés (lemme 5.15) et (c) $t' \geq t \times x$, chaque simulateur q_i peut exécuter les codes d'au moins $n - t \times x \geq (n - t')$ processus simulés jusqu'à ce qu'ils décident. Parce que l'algorithme exécuté par les processus simulés $p_1, \ldots, p_n$ est t'-résilient, ces $n - t \times x$ processus simulés décideront une valeur. Par conséquent, chaque simulateur calcule les valeurs de décision d'au moins $(n - t')$ processus simulés. □

Lemme 5.17. *Tous les simulateurs qui terminent la simulation de la k-ième invocation de l'opération mem.snapshot() par un processus simulé p_j obtiennent la même valeur pour cette invocation.*

Démonstration. La k-ième invocation de mem.snapshot() par un processus simulé p_j est simulée par une invocation sim_snapshot() correspondante effectuée par chaque simulateur q_i. À cause de la propriété d'accord de l'objet safe_agreement utilisé pour implémenter sim_snapshot(), la même valeur est renvoyée à chaque simulateur (lignes 05-06 de la figure 5.4). □

Lemme 5.18. *Soit $xcons[a]$ un objet x-consensus accédé par un ensemble de x processus simulés. Tous les simulateurs qui invoquent l'opération* sim_x_cons_propose$^{a}_{i,j}$() *associée aux invocations de $xcons[a]$.*x_cons_propose() *par les x processus (qui peuvent l'invoquer) obtiennent la même valeur.*

Démonstration. La démonstration est similaire à celle du lemme précédent. L'objet $XSAFE\_AG[a]$ de type safe_agreement, utilisé pour implémenter la simulation de $xcons[a]$, assure qu'au plus une valeur peut être décidée (lignes 02-03 de la figure 5.6). □

Lemme 5.19. *Pour chaque processus simulé p_j, les simulateurs décident au plus une valeur.*

Démonstration. Observons que les seules opérations non-déterministes qu'un processus simulé peut invoquer sont les opérations mem.snapshot() et (pour chaque a) les opérations $xcons[a]$.x_cons_propose(). Les lemmes 5.17 et 5.18 impliquent que tous les simulateurs obtiennent la même valeur quand ils invoquent les opérations $\mathsf{sim\_snapshot}_{i,j}()$ ou $\mathsf{sim\_x\_cons\_propose}^{a}_{i,j}()$ (qui simulent mem.snapshot() et $xcons[a]$.x_cons_propose() invoquées par p_j), ce qui démontre le lemme. □

Lemme 5.20. *Les séquences de $\mathsf{sim\_write}_{i,j}()$, $\mathsf{sim\_snapshot}_{i,j}()$ et $\mathsf{sim\_x\_cons\_propose}^{a}_{i,j}()$ invoquées par chaque simulateur (dans $ASM(n,t,1)$) pour le compte de chaque processus simulé p_j constituent une exécution correcte de l'algorithme simulé.*

Démonstration. Observons d'abord que, selon les lemmes 5.17 et 5.18, tous les simulateurs qui ne sont pas bloqués durant la simulation d'un processus p_j reçoivent la même valeur quand p_j invoque une opération non-déterministe. Ils simulent donc tous p_j de la même manière.

Quand un simulateur q_i exécute $\mathsf{sim\_snapshot}_{i,j}()$ (c'est-à-dire quand il simule une invocation de l'opération mem.snapshot() par p_j), il stocke dans sa variable locale $input_i$ les valeurs écrites par les simulateurs les plus avancés dans leur simulation pour chaque processus simulé (voir la figure 5.3 et les lignes 02-03 de la figure 5.4). Il ne peut choisir sa propre valeur de snapshot $input_i$ que si aucun autre processus n'a déjà terminé son exécution de ce e_sim_snapshot() (le lemme 5.17 implique que les objets safe_agreement ont un effet "mémoire"). Pour chaque e_sim_snapshot(), q_i renvoie donc une valeur de $input$ calculée par lui-même ou un autre simulateur. Remarquons que, à l'instant auquel cette valeur de $input$ a été déterminée, aucun simulateur n'avait terminé son e_sim_snapshot() correspondant. (Si ce n'était pas le cas, ce simulateur aurait fourni sa propre valeur de $input$ aux autres simulateurs.) Parce que les processus sont simulés de manière déterministe, la valeur de $input$ qui est renvoyée contient la dernière valeur écrite par p_j du point de vue de q_i. Cela montre que l'ordre de chaque processus simulé est respecté.

Pour garantir que la simulation est correcte, nous devons maintenant montrer que les opérations mem.write(), mem.snapshot() et $x\_cons[a]$.x_cons_propose() de tous les processus simulés peuvent être linéarisés. Pour cela, les points de linéarisation sont définis de la façon suivante.

- Le point de linéarisation d'un $mem[j]$.write() invoqué par p_j est placé à la ligne 03 de la figure 5.3 du premier simulateur qui exécute $\mathsf{sim\_write}_{i,j}()$.
- Le point de linéarisation d'un mem.snapshot() invoqué par p_j est placé à la ligne 01 de la figure 5.5 du simulateur q_i qui impose sa valeur de $input_i$ à l'objet safe_agreement correspondant.
- Le point de linéarisation d'un $x\_cons[a]$.x_cons_propose() invoqué par p_j est placé à la ligne 02 de la figure 5.6 du simulateur q_i dont l'invocation $XSAFE\_AG[a]$.$\mathsf{sa\_propose}_i()$ impose sa valeur à l'objet safe_agreement $XSAFE\_AG[a]$ correspondant.

Parce que le simulateur q_i qui impose sa valeur de $input_i$ dans une opération sim_snapshot() lit les valeurs les plus avancées au moment du snapshot (lignes 02-03 de la figure 5.5) et parce qu'une fois qu'un simulateur termine l'exécution de sim_snapshot(), la valeur pour ce sim_snapshot() ne peut plus changer (lemme 5.17), en ce qui concerne les opérations $mem[j]$.write() et mem.snapshot(), la linéarisation correspond à une linéarisation d'une exécution correcte de l'algorithme simulé. Un raisonnement similaire montre que la linéarisation est également correcte par rapport aux opérations $x\_cons[a]$.x_cons_propose() invoquées par les processus simulés. □

Théorème 5.21. *Soit $t \leq \lfloor \frac{t'}{x} \rfloor$. Les algorithmes décrits dans les figures 5.3, 5.4 et 5.6 sont une simulation correcte de $ASM(n,t',x)$ dans $ASM(n,t,1)$ (pour tout algorithme A qui résout une tâche de décision non colorée dans $ASM(n,t',x)$).*

Démonstration. La démonstration de la correction de la simulation est une conséquence directe du lemme 5.16 pour la vivacité, et des lemmes 5.19 et 5.20 pour la sûreté. □

5.2.3 Simulation de $ASM(n, t, 1)$ dans $ASM(n, t', x)$

5.2.3.1 Les objets partagés par les simulateurs

Soit A un algorithme qui résout une tâche non colorée dans $ASM(n, t, 1)$. Cela signifie que A est un algorithme t-résilient pour n processus et que ses processus p_1, ..., p_n coopèrent en accédant à une mémoire snapshot $mem[1..n]$ avec des opérations $mem[j]$.write() et mem.snapshot().

En supposant $t \geq \lfloor \frac{t'}{x} \rfloor$, cette section décrit une simulation de A dans le modèle de système $ASM(n, t', x)$, où un simulateur q_i peut en plus accéder à des objets x-consensus. Les simulateurs accèdent à une mémoire snapshot $MEM[1..n]$ comme dans la simulation précédente. La simulation de $mem[j]$.write() est exactement la même que celle décrite dans l'opération sim_write$_{i,j}$() de la figure 5.3.

Le cœur de cette simulation est la définition de l'opération sim_snapshot$_{i,j}$() exécutée par q_i pour simuler l'invocation de mem.snapshot() par p_j. La difficulté vient du fait que jusqu'à t' processus peuvent crasher dans $ASM(n, t', x)$, tandis que seuls t processus simulés sont autorisés à crasher dans $ASM(n, t, 1)$, et t' peut être plus grand que t.

5.2.3.2 Le type d'objet x_safe_agreement

Définition Ce type d'objet est une extension du type safe_agreement type décrit dans la section 5.1.2.1. Il est définit par deux opérations, notées x_sa_propose() et x_sa_decide() qu'un simulateur peut invoquer (au plus une fois et dans cet ordre). De plus, chaque objet x_safe_agreement considère que x des simulateurs sont ses *propriétaires* (ils sont définis dynamiquement et donc différents objets n'ont pas nécessairement le même ensemble de propriétaires). Les propriétés qui définissent le type x_safe_agreement sont les suivantes.

- *Terminaison*. Si au plus $(x - 1)$ processus crashent durant l'exécution de x_sa_propose(), chaque simulateur correct qui invoque x_sa_decide() termine cette invocation.
- *Accord*. Au plus une valeur est décidée.
- *Validité*. Une valeur décidée est une valeur proposée.

Comme nous le verrons, tandis que le cas $x = 1$ correspond à la définition du type safe_agreement, l'implémentation du type x_safe_agreement n'est pas une extension simple de l'algorithme décrit dans la figure 5.1.

L'opération sim_snapshot$_{i,j}$ Considérons la section 5.1.3.3 dans laquelle, au lieu d'objets de type safe_agreement, le tableau $SAFE\_AG[1..n, 1.. + \infty)$ contient des objets x_safe_agreement. L'algorithme sim_snapshot$_{i,j}$() invoqué par un simulateur q_i pour simuler l'invocation de mem.snapshot() par un processus simulé p_j est alors la même que celle décrite dans la figure 5.4 après avoir remplacé sa_propose() et sa_decide() par x_sa_propose() et x_sa_decide(), respectivement.

5.2.3.3 Implémentation du type x_safe_agreement

L'implémentation d'un objet x_safe_agreement est au cœur de la simulation. Elle se base sur des objets x-consensus (l'opération pour y accéder est notée x_cons_propose()), des registres atomiques multi-écrivains/multi-lecteurs (rappelons que ces registres peuvent être implémentés en utilisant une mémoire snapshot) et des objets Test&Set (qui peuvent être implémentés avec des objets x-consensus quand $x \geq 2$ [63]).

Associer dynamiquement des propriétaires à un objet x_safe_agreement Rappelons que, grâce aux opérations enter_mutex1 et exit_mutex1 qui sont invoquées avant et après $\mathsf{x\_sa\_propose}_i(v)$ (voir la figure 5.4), un simulateur q_i est engagé dans au plus un objet x_safe_agreement à la fois (en ce qui concerne l'opération $\mathsf{x\_sa\_propose}_i()$).

D'un autre coté, comme spécifié dans sa propriété de terminaison, un objet x_safe_agreement ne peut "crasher" (bloquer le processus simulé dont l'invocation de snapshot correspond à l'invocation $\mathsf{x\_sa\_decide}_i()$ de cet objet) que quand x des simulateurs qui l'accèdent crashent. Nous verrons que ce ne sera le cas que si ses x propriétaires crashent. Cela signifie que si tous les objets x_safe_agreement avaient le même ensemble de x propriétaires, contraindre un simulateur à n'être engagé que dans une seule opération $\mathsf{x\_sa\_propose}_i()$ à la fois ne serait pas suffisant car le crash des x propriétaires pourrait crasher tous les objets x_safe_agreement et la simulation pourrait rester bloquée indéfiniment. Pour empêcher ceci d'arriver, la simulation impose que les propriétaires d'un objet soient définis dynamiquement. Intuitivement, ce sont les x "premiers" simulateurs qui invoquent $\mathsf{x\_sa\_propose}_i()$ sur cet objet. Si t' simulateurs crashent, ils peuvent donc entrainer le crash d'au plus $\lfloor \frac{t'}{x} \rfloor$ objets x_safe_agreement, qui eux-mêmes peuvent bloquer au plus $\lfloor \frac{t'}{x} \rfloor$ processus simulés. D'où la contrainte $t \geq \lfloor \frac{t'}{x} \rfloor$.

```
operation x_compete_i() :
(01) ℓ ← 1 ; winner ← false ;
(02) while (ℓ ≤ x ∧ ¬winner) do
(03)       winner ← TS[ℓ].test&set() ; ℓ ← ℓ + 1
(04) end while ;
(05) return(winner).
```

FIGURE 5.7 – Une implémentation de l'opération $\mathsf{x\_compete}_i()$ (code de q_i)

Un objet noté $X\_T\&S$ est associé à chaque objet x_safe_agreement. Il fournit aux simulateurs une seule opération, notée $\mathsf{x\_compete}_i()$, qui renvoie $true$ à x simulateurs (si x processus ou moins l'invoquent, ceux qui ne crashent pas obtiennent tous $true$). Cette opération est implémentée par l'algorithme décrit dans la figure 5.7 qui utilise un tableau de x objets Test&Set. Un tel objet renvoie $true$ à la première invocation et $false$ aux invocations suivantes. Un objet Test&Set peut être implémenté à partir d'objets x-consensus si $x \geq 2$ [63].

Les simulateurs qui obtiennent $true$ quand ils invoquent $X\_T\&S.\mathsf{x\_compete}_i()$ sont les propriétaires de l'objet x_safe_agreement correspondant. On peut remarquer que, comme un simulateur peut invoquer $\mathsf{x\_sa\_propose}_i()$ sur au plus un objet x_safe_agreement à la fois, et comme un simulateur ne peut être propriétaire que de l'objet pour lequel il a une invocation $\mathsf{x\_sa\_propose}_i()$ en cours, aucun simulateur ne peut être le propriétaire de plusieurs objets x_safe_agreement au même moment.

L'opération $\mathsf{x\_sa\_decide}_i()$ La valeur décidée dans un objet x_safe_agreement est stockée dans un registre associé X_SAFE_AG (initialisé à $\perp$). Par conséquent, quand un simulateur q_i invoque $\mathsf{x\_sa\_decide}_i()$ sur cet objet x_safe_agreement, il attend que ce registre reçoive une valeur, puis renvoie cette valeur (lignes 09-10 de la figure 5.8).

L'opération $\mathsf{x\_sa\_propose}_i(v)$ L'algorithme qui implémente $\mathsf{x\_sa\_propose}_i(v)$ est décrit dans la figure 5.8. Un simulateur q_i commence par invoquer $X\_T\&S.\mathsf{x\_compete}_i()$ pour savoir si c'est un propriétaire de l'objet. Si ce n'est pas le cas, son invocation $\mathsf{x\_sa\_propose}_i()$ termine. Remarquons que, dans ce cas, au moins x simulateurs ont invoqué x_sa_propose() sur cet objet x_safe_agreement et x d'entre eux sont ses propriétaires (qui peuvent ne pas être corrects).

Si q_i est un propriétaire, il doit coopérer avec les autres propriétaires de ce objet x_safe_agreement de façon à ce qu'une des valeurs qu'ils proposent devienne la valeur décidée pour cet objet. Pour cela, ces simulateurs pourraient utiliser un objet x-consensus accessible seulement par ces x propriétaires. Le

```
operation x_sa_propose_i(v) :
(01) owner_i ← X_T&S.x_compete_i() ;
(02) if (owner_i) then
(03)    res ← v ;
(04)    for ℓ from 1 to m do
(05)       if (i ∈ SET_LIST[ℓ]) then res ← XCONS[ℓ].x_cons_propose(res) end if
(06)    end for ;
(07)    X_SAFE_AG ← res
(08) end if.

operation x_sa_decide_i() :
(09) wait (X_SAFE_AG ≠ ⊥) ;
(10) return(X_SAFE_AG).
```

FIGURE 5.8 – Une implementation du type x_safe_agreement (code de simulator q_i)

problème est qu'un simulateur ne sait pas qui sont les autres propriétaires et donc, ne sait pas quel est cet objet x-consensus.

Pour résoudre ce problème, deux tableaux sont associés à chaque objet x_safe_agreement. Soit m le nombre de sous-ensembles de taille x dans un ensemble de n éléments. Nous avons :

- $SET\_LIST[1..m]$ est un tableau qui contient les m sous-ensembles de simulateurs de taille x. $SET\_LIST[\ell]$ contient le sous-ensemble identifié par ℓ.
- $XCONS[1..m]$ est un tableau de m objets x-consensus. $XCONS[\ell]$ est un objet x-consensus qui peut être accédé par les simulateurs qui définissent le sous-ensemble $SET\_LIST[\ell]$ de taille x.

Si le simulateur q_i est un propriétaire, il parcourt la liste $SET\_LIST[1..m]$ (tous les propriétaires doivent la parcourir dans le même ordre). Quand il rencontre un ensemble $SET\_LIST[\ell]$ qui contient son identité i, q_i invoque x_cons_propose(res) sur l'objet $XCONS[\ell]$ correspondant et adopte la valeur qu'il obtient comme estimation de la valeur décidée dans l'objet x_safe_agreement. Remarquons que, quel que soit l'ensemble S de propriétaires de l'objet, q_i rencontre nécessairement S.

Quand il a fini son parcours de la liste, q_i dépose la valeur res dans X_SAFE_AG, qui est alors la valeur décidée dans cet objet x_safe_agreement.

5.2.3.4 Démonstration de la simulation de $ASM(n,t,1)$ dans $ASM(n,t',x)$

La démonstration consiste en deux théorèmes : (1) la démonstration que l'algorithme présenté dans la figure 5.8 implémente le type d'objet x_safe_agreement et (2) la démonstration de la simulation elle-même.

Théorème 5.22. *L'algorithme décrit dans la figure 5.8 implémente le type d'objet* x_safe_agreement

Démonstration. La propriété de validité (une valeur décidée est une valeur proposée) est une conséquence directe du texte de l'algorithme et de la propriété de validité des objets x-consensus $XCONS[1..m]$ utilisés.

$XSAG$ étant un objet x_safe_agreement, soit $owners[XSAG]$ son ensemble de simulateurs propriétaires. On peut remarquer que $|owners[XSAG]| \leq x$ (voir la figure 5.7).

Pour la propriété d'accord, nous devons montrer que les invocations $XSAG$.x_sa_decide() renvoient la même valeur. Pour cela, nous allons montrer que toutes les écritures du registre atomique X_SAFE_AG (utilisé pour implémenter $XSAG$) écrivent la même valeur v (ligne 07).

Selon la définition de $SET\_LIST[1..m]$, $\exists \ell$ tel que $owners[XSAG] \subseteq SET\_LIST[\ell]$. De plus, à cause de la propriété d'accord de l'objet x-consensus $XCONS[\ell]$, tous les simulateurs qui termine leurs invocations $XCONS[\ell]$.x_cons_propose() reçoivent la même valeur v. Pour tout $\ell' > \ell$, chaque

simulateur de $owners[XSAG]$ propose donc v à chaque objet $XCONS[\ell']$ auquel il accède et donc (comme seuls les simulateurs de $owners[XSAG]$ accèdent à ces objets), seule la valeur v peut être décidée dans ces objets. Chaque simulateur de $owners[XSAG]$ qui exécute la ligne 07 écrit donc v dans le registre atomique X_SAFE_AG, ce qui conclut la démonstration de la propriété d'accord de l'objet x_safe_agreement $XSAG$.

Pour démontrer la propriété de terminaison de $XSAG$, supposons que au plus $(x-1)$ simulateurs de $owners[XSAG]$ crashent durant l'exécution de $XSAG$.x_sa_propose(). Nous devons montrer que chaque simulateur correct q_i termine son invocation $XSAG$.x_sa_decide(). Comme au moins un simulateur (q_i) de $owners[XSAG]$ est correct, ce simulateur invoque $XSAG$.x_sa_propose(). Comme les objets $XCONS[\ell]$ accédés par q_i sont sans-attente, q_i finit par écrire une valeur différente de $\perp$ dans X_SAFE_AG. Aucun simulateur correct ne sera donc bloqué durant son exécution de l'opération $XSAG$.x_sa_decide(), ce qui conclut la démonstration du théorème. □

Démonstration de la simulation Comme précédemment, pour garantir la terminaison, un algorithme t-résilient nécessite qu'au plus t processus (simulés) crashent. Nous montrons ici que, quand au plus $t' \leq t \times x + (x-1)$ simulateurs utilisant des objets x-consensus crashent, (c'est-à-dire $t \geq \lfloor \frac{t'}{x} \rfloor$), chaque simulateur correct peut exécuter les codes d'au moins $(n-t)$ processus simulés sas être bloqués indéfiniment dans l'exécution de ces codes.

Pour qu'un processus simulé p_j décide la même valeur à chaque simulateur, les valeurs de snapshot renvoyées à p_j doivent être les mêmes pour tous les simulateurs et doivent être cohérentes avec les opérations d'écriture.

Lemme 5.23. *Soit $t \geq \lfloor \frac{t'}{x} \rfloor$. Les simulations d'au plus t processus simulés peut être bloquée indéfiniment (c'est-à-dire crashée) durant la simulation.*

Démonstration. Le seul endroit où la simulation d'un processus p_j peut être bloquée de façon permanente est quand p_j invoque mem.snapshot(), c'est-à-dire quand les simulateurs invoquent l'opération de simulation sim_snapshot() correspondante. Comme cette opération utilise un x_safe_agreement (invocation de x_sa_propose()), au moins x simulateurs doivent crasher durant l'exécution de l'opération x_sa_propose() correspondante pour que l'objet x_safe_agreement crashe et bloque de façon permanente le processus simulé p_j.

D'un autre coté, à cause (a) du mécanisme d'exclusion mutuelle utilisé dans sim_snapshot() et (b) de la détermination dynamique des x propriétaires d'un objet x_safe_agreement, le crash d'un simulateur ne peut participer qu'au crash d'un seul processus simulé p_j.

Donc, (a) il est nécessaire qu'au moins x simulateurs crashent pour bloquer indéfiniment un processus simulé et (b) ces x crashes ne peuvent pas bloquer d'autres processus simulés.

Comme au plus t' simulateurs peuvent crasher, au plus $\lfloor \frac{t'}{x} \rfloor \leq t$ processus simulés peuvent rester bloquer indéfiniment dans leur simulation, ce qui conclut la démonstration du lemme. □

Lemme 5.24. *Soit $t \geq \lfloor \frac{t'}{x} \rfloor$. Chaque simulateur correct q_i calcule les valeurs de décision d'au moins $(n-t)$ processus simulés.*

Démonstration. Le lemme 5.23 implique que chaque simulateur correct q_i simule entièrement les codes d'au moins $n - \lfloor \frac{t'}{x} \rfloor \geq n - t$ processus simulés. Comme l'algorithme simulé est t-résilient, la simulation de ces $n - \lfloor \frac{t'}{x} \rfloor$ processus simulés leur fournit à chacun une valeur de décision. Chaque simulateur correct q_i obtient donc des valeurs de décision pour au moins $(n-t)$ processus simulés. □

Lemme 5.25. *Les simulateurs qui terminent la simulation de la k-ième invocation de mem.snapshot() par un processus simulé p_j obtiennent tous la même valeur pour cette invocation de snapshot simulée.*

Démonstration. La démonstration est la même que celle du lemme 5.17 dans laquelle le type d'objet safe_agreement est remplacé par x_safe_agreement. (Remarquons que les deux démonstrations se basent uniquement sur la propriété d'accord de ces types d'objets.) □

Lemme 5.26. *Pour chaque processus simulé p_j, les simulateurs décident au plus une valeur.*

Démonstration. La démonstration est similaire à celle du lemme 5.19. Observons que les opérations mem.snapshot() sont les seules opérations non-déterministes invoquées par un processus p_j. Le lemme 5.25 implique que tous les simulateurs corrects q_i obtiennent la même valeur quand ils invoquent l'opération $\mathsf{sim\_snapshot}_{i,j}()$, ce qui démontre le lemme. □

Lemme 5.27. *Les séquences de* $\mathsf{sim\_write}_{i,j}()$ *et* $\mathsf{sim\_snapshot}_{i,j}()$ *invoquées par chaque simulateur (dans $ASM(n, t', x)$) pour le compte de chaque processus simulé p_j constituent une exécution correcte de l'algorithme simulé.*

Démonstration. La démonstration est la même que celle du lemme 5.20, dans laquelle la référence au lemme 5.17 est remplacée par une référence au lemme 5.25 et dans laquelle seules sont considérées les opérations de simulation sim_write() et sim_snapshot(). □

Théorème 5.28. *Soit $\lfloor \frac{t'}{x} \rfloor \leq t$. Les algorithmes décrits dans les figures 5.3, 5.4 (où* sa_propose() *et* sa_decide() *sont replacés par* x_sa_propose() *et* x_sa_decide()*) et 5.8 sont une simulation correcte du modèle $ASM(n, t, 1)$ dans $ASM(n, t', x)$ (pour tout algorithme qui résout une tâche de décision non colorée dans $ASM(n, t, 1)$).*

Démonstration. La démonstration est la conséquence du théorème 5.22 et des lemmes 5.24, 5.26 et 5.27. □

5.2.4 Généralisation

5.2.4.1 $ASM(n, t', x) \simeq ASM(t + 1, t, 1)$

Les simulations précédentes montrent que, quand $t = \lfloor \frac{t'}{x} \rfloor$, (c'est-à-dire $t \times x \leq t' \leq t \times x + (x - 1)$), tout algorithme qui résout une tâche non colorée dans $ASM(n, t, 1)$ peut être utilisé pour la résoudre dans $ASM(n, t', x)$ et vice-versa. D'un autre coté, la simulation BG montre que tout algorithme qui résout une tâche non colorée dans $ASM(n, t, 1)$ peut être utilisé pour la résoudre dans $ASM(t + 1, t, 1)$ et vice-versa.

Les observations précédentes impliquent que, quand $t = \lfloor \frac{t'}{x} \rfloor$, tout algorithme qui résout une tâche non colorée dans $ASM(n, t', x)$ peut être utilisé pour la résoudre dans $ASM(t + 1, t, 1)$ et vice-versa. Cette équivalence est notée $ASM(n, t', x) \simeq ASM(t + 1, t, 1)$.

5.2.4.2 Cas général : $ASM(n_1, t_1, x_1) \simeq ASM(n_2, t_2, x_2)$

En supposant $n_1 > t_1$ et $n_2 > t_2$, grâce à la simulation décrite dans les sections 5.2.2 et 5.2.3, nous avons $ASM(n_1, t_1, x_1) \simeq ASM(n_1, \lfloor \frac{t_1}{x_1} \rfloor, 1)$ et $ASM(n_2, t_2, x_2) \simeq ASM(n_2, \lfloor \frac{t_2}{x_2} \rfloor, 1)$.

Supposons que $\lfloor \frac{t_1}{x_1} \rfloor = \lfloor \frac{t_2}{x_2} \rfloor = t$. Grâce à la simulation BG, nous avons $ASM(n_1, t, 1) \simeq ASM(t + 1, t, 1)$ et $ASM(n_2, t, 1) \simeq ASM(t + 1, t, 1)$. Par transitivité, nous avons alors $ASM(n_1, t_1, x_1) \simeq ASM(n_2, t_2, x_2)$. Cette équivalence est illustrée dans la figure 5.9.

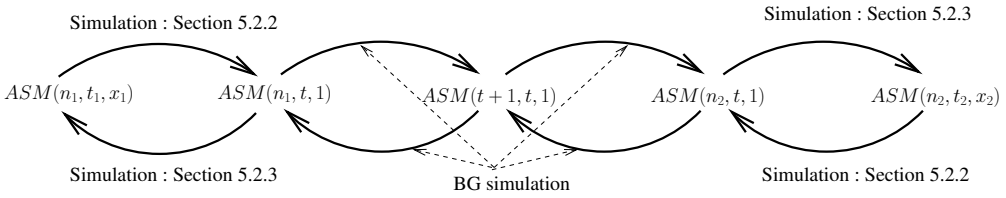

FIGURE 5.9 – Équivalence de modèles (pour $\lfloor \frac{t_1}{x_1} \rfloor = \lfloor \frac{t_2}{x_2} \rfloor$)

5.2.4.3 Augmenter la puissance des objets x-consensus peut être inutile

Modifier les paramètres du modèle Considérons les modèles $ASM(n, t, x)$ et $ASM(n, t, x + \Delta x)$. Selon les observations précédentes, même si le deuxième a des objets plus puissants que le premier, il n'est pas plus puissant si $\lfloor \frac{t}{x} \rfloor = \lfloor \frac{t}{x+\Delta x} \rfloor$. Des objets de base plus puissants n'augmentent donc pas obligatoirement la puissance du modèle. De façon similaire, augmenter le nombre de crashes possibles de t à $t + \Delta t$ n'affaiblit pas le modèle si $\lfloor \frac{t}{x} \rfloor = \lfloor \frac{t+\Delta t}{x} \rfloor$.
Cela montre la *puissance multiplicative du x-consensus*, qui établit que $ASM(n, t, 1) \simeq ASM(n, t', x)$ pour $t \times x \leq t' \leq (t \times x) + (x - 1)$.

Classes d'équivalence Plus généralement, cela montre que l'ensemble des modèles de systèmes peut être partitionné en classes d'équivalence. Tous les modèles $ASM(n, t', x)$ tels que $\lfloor \frac{t'}{x} \rfloor = t$ appartiennent à la même classe (du point de vue de la calculabilité des tâches non colorées) et $ASM(n, t, 1)$ peut être pris comme la forme canonique représentant l'ensemble des modèles de cette classe.

Comme exemple, fixons $t' = 8$. Nous avons :

- Tous les modèles de systèmes $ASM(n, 8, x)$, pour $9 \leq x \leq n$, ont la même puissance que $ASM(n, 0, 1)$.
- Les 4 modèles $ASM(n, 8, x)$, pour $5 \leq x \leq 8$, ont la même puissance que $ASM(n, 1, 1)$.
- Les modèles $ASM(n, 8, 4)$ et $ASM(n, 8, 3)$ ont la même puissance que $ASM(n, 2, 1)$.
- Le modèle $ASM(n, 8, 2)$ a la même puissance que $ASM(n, 4, 1)$.
- La dernière classe est $ASM(n, 8, 1)$.

Plus généralement, nous avons : si $\frac{t'}{t} \geq x > \frac{t'}{t+1}$ alors $ASM(n, t', x) \simeq ASM(n, t, 1)$. Cela signifie que, étant donnés t et t', tous les objets x_1-consensus, x_2-consensus, etc. qui satisfont les inégalités précédentes sont équivalents dans un système de n processus dans lequel jusqu'à t' processus peuvent crasher.

Une hiérarchie de modèles Comme indiqué dans le chapitre 2, Gafni et Kuznetsov ont introduit la notion de *nombre d'accord ensembliste* d'une tâche [58]. Le nombre d'accord ensembliste d'une tâche T est k si T ne peut pas être résolue de manière sans attente dans un système asynchrone dans lequel les processus ont accès à des registres et à des objets $(k + 1)$-accord ensembliste, mais peut être résolue de manière sans attente quand les processus ont accès à des objets k-accord ensembliste. Dans un système de n processus, les nombres d'accord ensembliste définissent une hiérarchie des tâches de taille n. Une tâche de la classe k est plus difficile qu'une tâche de la classe $(k + 1)$.

Le problème du k-accord ensembliste est impossible à résoudre dans $ASM(n, k, 1)$ [27, 75, 112], mais il peut être résolu dans $ASM(n, k - 1, 1)$ [37]. Toute tâche qui a le nombre d'accord ensembliste k peut donc être résolu dans $ASM(n, k - 1, 1)$ mais pas dans $ASM(n, k, 1)$. Ceci établit la hiérarchie suivante parmi les modèles de systèmes : Un modèle $\mathcal{S}$ est plus puissant qu'un modèle $\mathcal{S}'$ ($\mathcal{S} \succ \mathcal{S}'$), si plus de tâches peuvent être résolues dans $\mathcal{S}$ que dans $\mathcal{S}'$ (par exemple, $ASM(n, 3, 1) \succ ASM(n, 4, 1)$: le 4-accord ensembliste peut être résolu dans $ASM(n, 3, 1)$ mais pas dans $ASM(n, 4, 1)$).

La relation entre une tâche T_k, qui a le nombre d'accord ensembliste k, et un modèle $ASM(n, t, x)$ est alors la suivante : T_k peut être résolue dans $ASM(n, t, x)$ si et seulement si $k > \lfloor \frac{t}{x} \rfloor$. Parce que les

nombre d'accord ensembliste établissent une hiérarchie, la relation précédente établit une hiérarchie entre les modèles de système.

5.2.5 Conclusion

Cette section a traité la puissance de calculabilité des systèmes constitués de n processus qui communiquent en accédant à des registres et à des objets x-consensus avec $x \leq n$, et dont au maximum t peuvent crasher dans chaque exécution. Il a été montré que, en ce qui concerne les tâches non colorées, nous avons les équivalences suivantes :

- $ASM(n, t', x) \simeq ASM(n, t, 1)$ si et seulement si $\lfloor \frac{t'}{x} \rfloor = t$, c'est-à-dire $(t \times x) \leq t' \leq (t \times x) + (x - 1)$.
- $ASM(n, t, x) \simeq ASM(t + 1, t, x)$.
- $ASM(n_1, t_1, x_1) \simeq ASM(n_2, t_2, x_2)$ si et seulement si $\lfloor \frac{t_1}{x_1} \rfloor = \lfloor \frac{t_2}{x_2} \rfloor$.

Quand on considère la puissance de calculabilité d'un système asynchrone en mémoire partagée dans lequel les processus peuvent crasher, la simulation BG est donc un aspect du problème, tandis que la puissance multiplicative du x-consensus en est l'autre aspect.

Chapitre 6

Tâches de cassage de la symétrie

Une forme importante de coordination consiste à ce que les processus se distinguent les uns des autres. Cette forme de coordination est nécessaire pour "casser la symétrie" parmi des processus qui sont initialement dans le même état. Des formes spécifiques de cassage de la symétrie ont été étudiées par le passé, par exemple l'élection, l'exclusion mutuelle ou le renommage. Bien que les tâches de cassage de la symétrie soient cruciales dans certains problèmes d'informatique distribuée, elles ont été l'objet de moins d'attention que les tâches d'accord.

Renommage La tâche de cassage de la symétrie la plus connue et la plus étudiée est le renommage [17]. Dans ce problème, chacun des n processus qui constituent le système reçoit un identifiant unique pris parmi un domaine très large $[1..N]$ ($n << N$). Initialement, un processus ne connait que son identifiant, le nombre n de processus du système et le fait que tous les identifiants sont uniques. Les processus doivent coopérer afin de choisir un nouvel identifiant parmi un espace $[1..M]$ tel que $M << N$ et que deux processus n'aient pas le même nouvel identifiant. Étant donné M, le problème est appelé M-renommage.

Renommage adaptatif Dans certains cas, il est souhaitable que l'espace des nouveaux identifiants ne dépende que du nombre de processus p qui requièrent un nouvel identifiant, et pas du nombre total de processus du système. Dans ce cas, M ne dépend pas de n mais est une fonction de p. On appelle ce problème le renommage *adaptatif*. Plusieurs algorithmes de renommage adaptatif ont été conçus tels que $M = 2p - 1$ (par exemple [28]). Dans le cas du renommage adaptatif, $2p - 1$ est en fait une borne inférieure sur la taille de l'espace des nouveaux identifiants quand les processus doivent terminer de façon sans-attente et n'ont accès qu'à des registres. Une manière de contourner cette borne est d'utiliser des objets plus puissants [109]. Une autre manière est de restreindre la condition de progression.

Quand on considère la condition de progression sans-obstruction, il est possible de résoudre le consensus. De la même manière, il est possible de résoudre le renommage adaptatif optimal (c'est-à-dire $M = p$) sous cette condition. D'où la question suivante : "quelle est la plus petite valeur $M(k)$ qui peut être atteinte pour le $M(k)$-renommage adaptatif quand on considère la condition de progression k-sans-obstruction ?"

La première section de ce chapitre répond à cette question en présentant un algorithme de $M(k)$-renommage adaptatif qui satisfait cette condition de progression et tel que $M(k) = \min(2p-1, p+k-1)$. Cela signifie que, dans toutes les exécutions où le degré de concurrence finit par être limité à k, tous les processus terminent. Une démonstration que $M(k) = \min(2p - 1, p + k - 1)$ est une borne inférieure pour le $M(k)$-renommage adaptatif k-sans-obstruction est également fournie.

Tâches de cassage de la symétrie généralisé La deuxième section introduit les *tâches de cassage de symétrie généralisé* ou *tâches GSB* (pour *Generalized Symmetry Breaking*), une famille de tâches qui inclut l'élection [113], le renommage [17], le cassage de symétrie faible (*Weak Symmetry Breaking* ou

WSB) et de nombreux autres problèmes qui, auparavant, étaient considérés comme indépendants. La notion de tâche GSB permet donc d'unifier tous ces problèmes dans un seul cadre d'étude.

Une tâche GSB pour n processus est définie par un ensemble de valeurs de sortie possibles et, pour chaque valeur v, une borne inférieure et une borne supérieure sur le nombre de processus qui peuvent la décider.

6.1 Renommage adaptatif x-sans obstruction

Définition Le problème du renommage a été décrit dans la section 2. Chaque processus p_i commence avec un identifiant initial unique id_i. Ces identifiants initiaux appartiennent à un ensemble $\{1, \ldots, N\}$ tel que $n << N$. Soit new_name() la seule opération fournie aux processus par un objet de M-renommage adaptatif, c'est-à-dire un objet qui permet aux processus d'obtenir un nouvel identifiant unique dans intervalle $[1..M]$. Les propriétés de sûreté d'un tel objet (qui définissent le problème du M-renommage adaptatif) sont les suivantes. Dans ces propriétés, p désigne le nombre de processus qui invoquent l'opération new_name() durant une exécution (le nombre de *participants*).

- *Accord*. Chaque valeur est décidé par au plus un processus.
- *Adaptativité*. Une valeur décidée appartient à $[1..M]$ où M est une fonction de p.
- *Indépendance de l'index*. Le comportement d'un processus est indépendant de son index.

La dernière propriété établit que si, dans une exécution, un processus dont l'index est i obtient la valeur v, ce processus aurait obtenu la même valeur si son index était $j \neq i$. Cela signifie que, d'un point de vue opérationnel, les indexes ne peuvent servir qu'à des mécanismes d'adressage, par exemple pour accéder à un tableau. Les indexes ne peuvent pas être utilisés pour *calculer* de nouveaux identifiants.

Terminaison k-sans-obstruction Quand on considère la terminaison k-sans-obstruction, il est possible qu'aucun processus ne termine si aucun ensemble P d'au plus k processus ne s'exécute en isolation durant suffisamment longtemps. La propriété de terminaison k-sans-obstruction est définie de la façon suivante.

- *Terminaison k-sans-obstruction*. Pour tout sous-ensemble P de processus corrects, avec $|P| \leq k$, si les processus de P s'exécutent en isolation durant suffisamment longtemps, alors tous les processus de P terminent.

"En isolation" signifie que, après un instant τ, les processus de P sont les seuls à exécuter des instructions. Remarquons que des processus en dehors de P peuvent avoir été actifs avant l'instant τ.

6.1.1 Un algorithme de renommage adaptatif k-sans-obstruction

Objets partagés utilisés Les processus coopèrent en utilisant deux tableaux de registres atomiques mono-écrivain/multi-lecteurs notés $OLDNAMES[1..n]$, et $LEVEL[1..n]$ et un objet snapshot noté $NAMES$.

- Le registre $OLDNAMES[i]$, initialisé à $\perp$, est utilisé par le processus p_i pour stocker son identité id_i. $OLDNAMES[i] \neq \perp$ signifie donc que p_i participe au renommage.
- Le registre $LEVEL[i]$ est initialisé à 0. Pour obtenir un nouveau nom, les processus progressent de manière asynchrones d'un niveau (en commençant par 1) vers le suivant. $LEVEL[i]$ contient le niveau atteint par p_i. Comme nous le verrons, si durant un temps suffisamment long au plus k processus s'exécutent, ils se stabiliseront au même niveau et obtiendront de nouveaux noms.
- $NAMES[1..n]$ est un objet snapshot initialisé à $[\perp, \ldots, \perp]$. $NAMES[i]$ contient le nouveau nom que p_i essaye d'acquérir. Quand p_i termine son opération new_name(), $NAMES[i]$ contient son nouveau nom.

Comportement des processus L'algorithme 1 décrit le comportement d'un processus p_i. Quand il invoque new_name(id_i), p_i dépose son identité id_i dans $OLDNAMES[i]$ et avance du niveau 0 au niveau 1. La variable locale $prop_i$ contient la proposition de p_i pour son nouveau nom. Sa valeur initiale est $\perp$. Ensuite, p_i entre dans une boucle (lignes 03-21) dont il sortira à la ligne 06 avec son nouveau nom.

Chaque fois qu'il démarre une nouvelle exécution de la boucle, p_i écrit d'abord sa proposition de nouveau nom dans $NAMES[i]$ et lit (avec une invocation $NAMES$.snapshot()) les valeurs de toutes les propositions (ligne 04). Si sa proposition $prop_i$ n'est pas $\perp$ et si aucun autre processus n'a proposé le même nouveau nom (line 05), p_i définit son nouveau nom comme la valeur de $prop_i$ et sort de la boucle (ligne 06). Sinon, il y a un conflit : plusieurs processus essayent d'acquérir le même nouveau nom $prop_i$. Dans ce cas, p_i exécute les lignes 08-19 pour résoudre ce conflit. Ces lignes constituent le cœur de l'algorithme.

```
operation new_name(id_i) :
(01) prop_i ← ⊥ ; my_level_i ← 1 ;
(02) OLDNAMES[i] ← id_i ; LEVEL[i] ← my_level_i ;
(03) repeat forever
(04)   NAMES[i] ← prop_i ; names_i ← NAMES.snapshot() ;
(05)   if ((prop_i ≠ ⊥) ∧ (∀j ≠ i : names_i[j] ≠ prop_i))
(06)     then return(prop_i)
(07)     else levels_i ← LEVEL[1..n] ; highest_level_i ← max({levels_i[j]}) ;
(08)          if (my_level_i < highest_level_i)
(09)            then my_level_i ← highest_level_i ; LEVEL[i] ← my_level_i
(10)            else oldnames_i ← OLDNAMES[1..n] ;
(11)                 contending_i ← {oldnames_i[j] | levels_i[j] = highest_level_i} ;
(12)                 if (|contending_i| > k)
(13)                   then my_level_i ← highest_level_i + 1 ; LEVEL[i] ← my_level_i
(14)                   else let X    = {names_i[j] | names_i[j] ≠ ⊥} ;
(15)                        let free = the increasing sequence 1, 2, 3, ... from which
                                       the integers in X have been suppressed ;
(16)                        let r    = rank of id_i in contending_i ;
(17)                        prop_i   ← the rth integer in the increasing sequence free
(18)                 end if
(19)          end if
(20)   end if
(21) end repeat.
```

Algorithm 1: M-renommage adaptatif k-sans-obstruction avec $M = \min(2p - 1, p + k - 1)$

En cas de conflit, p_i lit d'abord, de manière asynchrone, toutes les entrées de $LEVEL[1..n]$ puis calcule le plus haut niveau atteint (ligne 07) $highest\_level_i$. Si son niveau est plus petit que $highest\_level_i$, p_i saute à ce niveau, l'indique en écrivant dans $LEVEL[i]$ (lignes 08-09) et passe à la prochaine itération de la boucle.

Si son niveau actuel est égal à $highest\_level_i$, p_i calcule l'ensemble des processus avec lesquels il est en concurrence pour acquérir un nouveau nom, l'ensemble $contending_i$ (lignes 10-11). Ce sont les processus dont le niveau est égal à $highest\_level_i$. Le comportement de p_i dépend ensuite de la taille de l'ensemble $contending_i$ (prédicat à la ligne 12).

- Si $|contending_i| > k$, il y a trop de processus en compétition quand on considère la propriété k-sans-obstruction. Le processus p_i progresse alors au prochain niveau et passe à la prochaine itération de la boucle (ligne 13).
- Si $|contending_i| \leq k$, p_i sélectionne une nouvelle proposition de nom avant de passer à la prochaine itération de la boucle. Cette sélection est similaire à ce qui est fait dans d'autres algorithmes de renommage (par exemple [25]). La séquence *free*, définie aux lignes 14-15, représente la liste des noms disponibles. Le processus p_i définit donc sa nouvelle proposition comme la $r$ième valeur

libre dans la liste *free* où r est son rang dans l'ensemble de (au plus k) processus en compétition (donc $1 \leq r \leq k$).

6.1.2 Démonstration de l'algorithme

Lemme 6.1. *Un nom ne peut pas être décidé par plus d'un processus.*

Démonstration. Supposons, pour arriver à une contradiction, que deux processus p_i et p_j décident le même nom nm (ils le décident à la ligne 06). Les opérations d'écriture et de snapshot à la ligne 04 et le prédicat correspondant de la ligne 05 impliquent que $names_i[i] = nm$ et $names_i[j] \neq nm$ quand p_i vérifie le prédicat à la ligne 05. De la même manière, comme p_j décide nm, nous avons $names_j[i] = nm$ et $names_j[i] \neq nm$ quand il exécute la ligne 05. De plus, ni p_i ni p_j ne modifie $NAMES$ après avoir décider.

D'un autre coté, comme toutes les opérations de l'objet snapshot $NAMES$ sont linéarisables, les dernières invocations effectuées par p_i et par p_j peuvent être ordonnées selon un ordre total. Supposons, sans perte de généralité, que l'invocation du snapshot par p_i précède celle de p_j. Nous avons donc $NAMES[i] = nm$ quand p_j invoque $NAMES$.snapshot(). Le tableau $names_j$ obtenu par p_j à la ligne 04 est donc tel que $names_j[i] = nm$ et $names_j[j] = nm$. Par conséquent, le prédicat vérifié par p_j à la ligne 05 est faux, ce qui contredit l'hypothèse initiale et conclut la démonstration du lemme. □

Lemme 6.2. *Soit p le nombre de processus qui participent au renommage. La taille du nouvel espace des noms est $M = \min(2p - 1, p + x - 1)$.*

Démonstration. Considérons une exécution dans laquelle au plus p processus participent. Soit p_i un processus qui renvoie un nouveau nom (ligne 06). Le texte de l'algorithme implique que ce nouveau nom a été obtenu par p_i à la ligne 17.

À cause de la définition même de la valeur p, quand p_i définit sa dernière proposition de nom, au plus $p - 1$ autres processus ont déjà définit une proposition. De plus, parce qu'il propose un nouveau nom seulement si $|contending| \leq k$ (ligne 12), le rang r de p_i dans l'ensemble $contending$ (ligne 16) est tel que $r \leq \min(p, k)$. La dernière proposition de nom calculée par p_i à partir de la paire $(free, r)$ (lignes 14-17) est donc bornée par $(p - 1) + \min(p, k)$, et donc $M = \min(2p - 1, p + k - 1)$. □

Lemme 6.3. *Pour chaque sous-ensemble P de processus tel que $|P| \leq k$, si les processus de P s'exécutent en isolation durant suffisamment longtemps, alors tous les processus de P exécutent la ligne 06 (ils décident un nouveau nom et terminent).*

Démonstration. Supposons, pour arriver à une contradiction, qu'il y a une exécution infinie de l'algorithme dans laquelle, après un temps fini τ, tous les processus de P exécutent un nombre infini d'étapes et aucun processus hors de P n'exécute d'étapes.

Parce que $|P| \leq k$ et parce que tous les processus de P exécutent un nombre infini d'étapes, il y a un temps fini $\tau' \geq \tau$ à partir duquel chaque processus p_j de P a affecté à $LEVEL[j]$ la valeur $\max(\{LEVEL[j]\}_{1 \leq j \leq n})$ (lignes 07-09) et ne modifie plus $LEVEL[j]$ (test à la ligne 08). Cela dû au fait que le test à la ligne 08 est alors toujours faux et donc, la ligne 09 n'est plus exécutée. $LEVEL$ ne change donc plus après τ' et tous les processus de P finissent par obtenir le même ensemble $contending$ (ligne 11). Chaque processus de P obtient donc un rang différent dans $contending$.

Soit p_i le processus de P qui a la plus petite identité id_i et r son rang dans $contending$ (ce rang n'est pas nécessairement 1 car il peut y avoir des processus crashés qui sont hors de P mais qui sont dans $contending$). Soit z le $r$ième entier de la séquence $free$ qui n'est pas pris par un processus hors de P P (lignes 15-18). Parce que, après τ', $LEVEL$ ne change plus et parce que p_i a le plus petit rang parmi les processus de P, une fois que tous les processus de P ont exécuté les lignes 15-18 après τ', p_i sera le seul processus à proposer $prop_i = z$ (tous les autres processus de P proposeront des noms plus grands). Le

prédicat évalué par p_i à la ligne 05 finit donc par être satisfait et, par conséquent, p_i exécute la ligne 06, ce qui contredit l'hypothèse initiale et complète la démonstration. □

Théorème 6.4. *L'algorithme 1 est un algorithme de M-renommage adaptatif k-sans-obstruction avec $M = \min(2p - 1, p + k - 1)$.*

Démonstration. La propriété d'accord vient du lemme 6.1. La propriété d'adaptativité vient du lemme 6.2. La propriété d'indépendance de l'index est une conséquence directe du texte de l'algorithme (les indexes sont utilisés uniquement pour l'adressage des entrées des tableaux).

La propriété de terminaison k-sans-obstruction requiert que, pour chaque sous-ensemble P de processus tel que (1) $|P| \leq k$ et (2) les processus de P s'exécutent en isolation durant suffisamment longtemps, les processus de P décident un nouveau nom. C'est précisément la propriété démontrée par le lemme 6.3. □

6.1.3 Résultats d'impossibilité et d'optimalité

Castañeda et Rajsbaum [32] ont montré que, pour le M-renommage non-adaptatif, il y a des nombres n *exceptionnels* de processus pour lesquels la borne inférieur sur la taille M de l'espace des nouveau noms est $M = 2n - 2$ (tandis que $M = 2n - 1$ pour les autres valeurs de n). Ces valeurs exceptionnelles de n correspondent aux cas dans lesquels l'ensemble $\{\binom{n}{i} \ : \ 1 \leq i \leq \lfloor \frac{n}{2} \rfloor\}$ est premier [32].

Le $(2n - 2)$-renommage adaptatif permet lui de résoudre le $(n - 1)$-accord ensembliste quand $t = n - 1$ [59]. L'impossibilité de résoudre le $(n - 1)$-accord ensembliste de manière sans-attente dans un système uniquement doté de registres a été montré pour n'importe quel nombre de processus [28, 75, 112]. La borne inférieure de $2p - 1$ sur la taille de l'espace des noms pour le renommage adaptatif sans-attente tient donc toujours. Cette borne inférieure est utilisée dans la démonstration du théorème suivant.

Théorème 6.5. *Soit p le nombre de processus qui participent au renommage. Il n'existe pas d'algorithme de M-renommage adaptatif k-sans-obstruction avec $M < \min(2p - 1, p + k - 1)$.*

Démonstration. Il y a deux cas : $p \leq k$ et $p > k$.

Premier cas : $p \leq k$. Quand moins de k processus participent, la terminaison k-sans-obstruction est équivalente à la terminaison sans-attente, et donc ce cas se réduit au renommage adaptatif sans-attente. La taille minimale de l'espace des noms est alors $2p - 1 = \min(2p - 1, p + k - 1)$.

Deuxième cas : $p > k$. Considérons une exécution dans laquelle $p' = p - k$ processus corrects participent et décident des nouveaux noms de manière non concurrente (l'un après l'autre). La condition de progression k-sans-obstruction impose que tous ces processus terminent leurs invocations. Ces p' processus occuperont alors p' noms qui ne pourront plus être décidés par des processus qui invoquerait une opération de renommage plus tard. Après que ces p' processus aient terminé leurs invocations, $k' \leq k$ des k processus restants invoquent le renommage et s'exécutent de manière concurrente. Parce que seuls $k' \leq k$ processus s'exécutent de manière concurrente, la terminaison k-sans-obstruction requiert que tous ces processus terminent. Ces k' processus auront alors besoin d'un nouvel espace de $2k' - 1 \leq 2k - 1$ noms (sans compter ceux des p' premiers processus) pour obtenir de nouveaux noms. L'espace des noms total devra alors être d'une taille d'au moins $p' + 2k - 1 = p + k - 1 = \min(2p - 1, p + k - 1)$, ce qui conclut la démonstration du théorème. □

Le corollaire suivant est une conséquence immédiate de l'algorithme 1 (suffisance) et du théorème précédent (nécessité).

Corollaire 6.6. *L'algorithme 1 est optimal par rapport à la taille de l'espace des noms.*

6.2 Tâches de cassage de symétrie généralisé

6.2.1 La famille des tâches de cassage de la symétrie généralisé (tâches GSB)

Définition et propriétés Comme dans la section précédente, on suppose que, dans chaque exécution, les processus démarrent avec un identifiant unique compris entre 1 et N. Une tâche de cassage de la symétrie généralisé (*Generalized Symmetry Breaking (GSB) task*) pour n processus, $\langle n, m, \vec{\ell}, \vec{u}\rangle$-GSB, $\vec{\ell} = [\ell_1, \ldots, \ell_m]$, $\vec{u} = [u_1, \ldots, u_m]$, est définie par les propriétés suivantes. Les paramètres n, m, $\vec{\ell}$ et $\vec{u}$ sont définis de façon statique. Cela signifie que les tâches GSB ne sont pas adaptatives.

- *Validité*. Une valeur décidée appartient à $[1..m]$.
- *Indépendance de l'index*. Le comportement d'un processus est indépendant de son index.
- *Accord asymétrique*. Si tous les processus décident une valeur, chaque valeur $v \in [1..m]$ est décidée par au moins ℓ_v et au plus u_v processus.

Quand toutes les bornes inférieures ℓ_v sont égales à une valeur ℓ et toutes les bornes supérieures égales à une valeur u, la tâche est une tâche GSB *symétrique* et est notée $\langle n, m, \ell, u\rangle$-GSB. La propriété d'accord asymétrique est alors remplacée par :

- *Accord symétrique*. Si tous les processus décident une valeur, chaque valeur $v \in [1..m]$ est décidée par au moins ℓ et au plus u processus.

Remarquons que les propriétés d'accord ne concernent que les exécutions dans lesquelles tous les processus décident. Ceci est dû au fait que, dans un système asynchrone (cas dans lequel nous nous plaçons pour cette étude) toute exécution finie dans laquelle une partie des processus crashent peut être complétée pour former une exécution dans laquelle tous les processus sont corrects. Ces propriétés sont donc également contraignantes pour les exécutions dans lesquelles tous les processus ne sont pas corrects.

Pour définir formellement une tâche GSB, soit $\mathcal{I}_N$ l'ensemble des vecteurs de dimension n ayant des entrées distinctes dans $1, \ldots N$. De plus, étant donné un vecteur V, soit $\#_x(V)$ le nombre d'entrées de V égales à x.

Définition 6.7 (Tâche GSB)**.** *Pour m, $\vec{\ell}$ et $\vec{u}$, la tâche $\langle n, m, \vec{\ell}, \vec{u}\rangle$-GSB est la tâche $(\mathcal{I}_N, \mathcal{O}, \Delta)$, où $\mathcal{O}$ consiste en l'ensemble des vecteurs O tels que $\forall\, v \in [1..m] :\ \ell_v \leq \#_v(O) \leq u_v$, et pour chaque $I \in \mathcal{I}_N$, $\Delta(I) = \mathcal{O}$.*

Nous disons qu'une tâche GSB est *faisable* si $\mathcal{O}$ n'est pas vide.

Lemme 6.8. *Une tâche GSB est faisable si et seulement si $\sum_{v=1}^{m} \ell_v \leq n \leq \sum_{v=1}^{m} u_v$.*

Démonstration. La démonstration est la conséquence directe de la propriété d'accord asymétrique. □

Pour le cas des tâches GSB symétriques, le lemme précédent peut être reformulé de la façon suivante.

Lemme 6.9. *Si $\forall\, v \in [1..m] :\ \ell_v = \ell$ et $\forall\, v \in [1..m] :\ u_v = u$, la tâche GSB est faisable si et seulement si $m \times \ell \leq n \leq m \times u$.*

Un algorithme est *basé sur des comparaisons* si les processus n'utilisent que des opérations de comparaison sur leurs entrées. Le lemme suivant généralise une propriété bien connue (citons par exemple [33, 35]) du renommage et du cassage de symétrie faible. Elle établit qu'on peut considérer sans perte de généralité qu'un algorithme qui résout une tâche GSB est basé sur des comparaisons. Cela peut être utile pour prouver des résultats d'impossibilité (par exemple [24, 32]).

Théorème 6.10. *Considérons une tâche $\langle n, m, \vec{\ell}, \vec{u}\rangle$-GSB, $T = (\mathcal{I}, \mathcal{O}, \Delta)$. Il existe un algorithme pour T dans un modèle M doté de registres si et seulement si il existe un algorithme basé sur des comparaisons pour T dans le modèle M.*

Démonstration. Supposons qu'il existe un algorithme $\mathcal{A}$ qui résout T dans le modèle M. Pour avoir un algorithme basé sur des comparaisons, les processus commencent par obtenir de nouvelles identités temporaires en utilisant n'importe quel algorithme de $(2n-1)$-renommage sans-attente utilisant des registres, par exemple celui de [25], en l'exécutant en utilisant leurs identités initiales de $\mathcal{I}$. Les identités intermédiaires appartiendront alors également à $\mathcal{I}$. Les processus peuvent alors utiliser ces identités pour exécuter $\mathcal{A}$, et l'algorithme qui en résulte est basé sur des comparaisons. L'autre direction est triviale. □

6.2.1.1 Instances de tâches de cassage de la symétrie généralisé

Rappelons que les paramètres n, m, $\vec{\ell}$ et $\vec{u}$ qui définissent une tâche GSB sont définis statiquement.

Élection Nous pouvons définir la tâche GSB asymétrique *élection* en établissant qu'exactement un processus doit décider 1 et exactement $n-1$ processus doivent décider 2.

L'élection est une tâche GSB avec accord asymétrique. Dans cette section, nous considérons principalement des tâches GSB avec accord symétrique Cela signifie que les m valeurs sont égales par rapport à la décision. Si dans une exécution correcte r, v est décidée par x processus et w est décidée par y processus, alors l'exécution dans laquelle v est décidée par y processus et w est décidée par x processus (et toutes les autres valeurs sont décidées comme dans r) est une exécution correcte. Les exemples suivants sont des tâches GSB symétriques.

k-cassage de symétrie faible $k \leq n/2$ (k-WSB) C'est la tâche $\langle n, 2, k, n-k\rangle$-GSB. Elle a une formulation simple. Un processus doit décider une valeur parmi deux possibles, et chaque valeur doit être décidée par au moins k et au plus $n-k$ processus. Remarquons que la tâche 1-WSB est la tâche connue de *cassage de symétrie faible (Weak Symmetry Breaking, WSB)*.

m-renommage Dans la tâche du m-renommage, mentionnée précédemment à plusieurs reprises, les processus doivent décider des nouvelles identités distinctes parmi l'ensemble $[1..m]$. On peut remarquer que le m-renommage est en fait la tâche $\langle n, m, 0, 1\rangle$-GSB [1].

Renommage parfait La tâche du *renommage parfait* est la tâche du m-renommage dans laquelle la taille m du nouvel espace des identifiants est "optimal", c'est-à-dire qu'il n'y a pas de solution avec $m' < m$ quel que soit le modèle utilisé. Cela signifie que $m = n$. Cette tâche est la tâche $\langle n, n, 1, 1\rangle$-GSB. Nous verrons que la tâche du renommage parfait est universelle parmi les tâches GSB.

k-Slot C'est une nouvelle tâche, définie de la façon suivante. Chaque processus doit décider une valeur dans $[1..k]$ et chaque valeur doit être décidée au mois une fois. C'est la tâche $\langle n, k, 1, n\rangle$-GSB, ou son synonyme, la tâche $\langle n, k, 1, n-k+1\rangle$-GSB. On peut remarquer que la tâche WSB (cassage de symétrie faible) est la tâche 2-slot.

6.2.2 La structure des tâches GSB symétriques

Cette section étudie la structure combinatoire des tâches GSB symétriques pour analyser les deux problèmes suivants : les synonymes et les inclusions de vecteurs de sortie. Cette analyse n'est pas distribuée. Les problèmes de calculabilité distribuée sont traités dans la section 6.2.3.

Remarquons que les tâches $G_1 = \langle n, m, \vec{\ell}_1, \vec{u}_1\rangle$-GSB et $G_2 = \langle n, m, \vec{\ell}_2, \vec{u}_2\rangle$-GSB peuvent en fait être la même tâche T (c'est-à-dire qu'elles ont le même ensemble de vecteurs de sortie). Dans ce cas,

1. Le problème du m-renommage *adaptatif*, dans lequel m dépend du nombre p de participants, n'est pas une tâche GSB.

nous écrivons $G_1 \equiv G_2$ et disons que G_1 et G_2 sont *synonymes*. Par exemple, $\langle n, 2, 1, n-1\rangle$-GSB, $\langle n, 2, 0, n-1\rangle$-GSB et $\langle n, 2, 1, n\rangle$-GSB sont synonymes.

Si l'ensemble $S(T_1)$ des vecteurs de sortie d'une tâche GSB T_1 est contenu dans l'ensemble $S(T_2)$ des vecteurs de sortie d'une tâche GSB T_2, il est clair que T_2 ne peut pas être plus difficile à résoudre que T_1. Comme $S(T_1) \subset S(T_2)$, tout algorithme qui résout T_1 résout également T_2. Dans ce cas, nous écrivons $T_1 \subset T_2$.

6.2.2.1 Vecteurs quantités and vecteurs noyaux associés à une tâche

Soit T une tâche $\langle n, m, \ell, u\rangle$-GSB définie par son ensemble de vecteurs de sortie $S(T)$. Nous associions à T un ensemble de vecteurs (appelés *vecteurs quantité* et *vecteurs noyaux*) définis de la manière suivante.

Définition 6.11. *Soit $O \in S(T)$. Le* vecteur quantité *V associé à O est le vecteur de dimension m tel que $\forall\, v \in [1..m]$: $V[v] = \#_v(O)$. Soit $C(T)$ l'ensemble des* vecteurs quantité *associés à T.*

Parce que nous considérons l'accord symétrique, les vecteurs quantité qui contiennent les mêmes valeurs (par exemple $[a, b, c]$, $[b, c, a]$ et $[c, a, b]$ quand on considère le cas $m = 3$) peuvent être représentés par un seul vecteur quantité $K[1..m]$, à savoir le vecteur dont chaque entrée est égale ou supérieure à la suivante (par exemple le vecteur quantité $[b, c, a]$ si $b \geq c \geq a$). Un tel vecteur représente tous les vecteurs de sortie de $S(T)$ dans lesquels la valeur la plus fréquente apparait $K[1]$ fois, la deuxième plus fréquente $K[2]$ fois, etc.

Définition 6.12. *Partitionnons $C(T)$ en ensembles X de vecteurs quantité tels que chaque ensemble X contient tous les vecteurs quantité qui sont des permutations de l'un d'entre eux.*

- *Le* vecteur noyau *de X est son vecteur quantité K tel que $K[1] \geq K[2] \geq \cdots \geq K[m]$.*
- *L'*ensemble noyau *de T est l'ensemble de tous ses vecteurs noyaux.*
- *Le* vecteur noyau central *de T est son vecteur noyau tel que $[\frac{n}{m}, \cdots, \frac{n}{m}]$ si n est un multiple de m, et $K = [\lceil \frac{n}{m} \rceil, \cdots, \lfloor \frac{n}{m} \rfloor]$ (avec les premières $n \mod m$ entrées égales à $\lceil \frac{n}{m} \rceil$) si n n'est pas un multiple de m.*

Le lemme suivant est une conséquence directe des définitions de *vecteur noyau* et *ensemble noyau*.

Lemme 6.13. *Étant donnée une tâche GSB T, son ensemble noyau est totalement ordonné par l'ordre lexicographique.*

Pour résumer,

- L'ensemble des tâches $\langle n, -, -, -\rangle$ GSB est ordonné partiellement (selon la relation d'inclusion sur les ensembles noyaux qui définissent les tâches),
- Si $T1 \subset T2$, tout vecteur de sortie (solution) de $T1$ est un vecteur de sortie (solution) de $T2$, d'où on en conclut que tout algorithme qui résout $T1$ résout également $T2$.

Exemples Toutes les tâches $\langle n, m, \ell, u\rangle$-GSB qui sont faisables avec $n = 6$, $m = 3$ et $u \leq n = 6$ sont décrites dans la table 6.1. Les 6 processus peuvent donc décider jusqu'à 3 valeurs différentes. Les vecteurs noyaux de chacune de ces tâches sont indiqués, et ces vecteurs noyaux sont ordonnés selon l'ordre lexicographique, de gauche à droite.

Comme exemple, le vecteur noyau $[4, 2, 0]$ représente tous les vecteurs de sortie dans lesquels la valeur la plus fréquente (qui est 1, 2 ou 3) apparait 4 fois, la deuxième plus fréquente apparait 2 fois et la troisième valeur possible n'apparait pas. Pour prendre un autre exemple, l'ensemble noyau de la tâche $\langle 6, 3, 0, 4\rangle$-GSB est constitué de cinq vecteurs noyaux : $\{[4, 2, 0], [4, 1, 1], [3, 3, 0], [3, 2, 1], [2, 2, 2]\}$. Observons finalement que le vecteur noyau central $[2, 2, 2]$ appartient à toutes les tâches. De plus, les

tâches GSB $\langle 6,3,2,5\rangle$, $\langle 6,3,2,4\rangle$, $\langle 6,3,2,3\rangle$, $\langle 6,3,0,2\rangle$, $\langle 6,3,1,2\rangle$ et $\langle 6,3,2,2\rangle$ sont synonymes. Les tâches GSB $\langle 6,3,1,6\rangle$, $\langle 6,3,1,5\rangle$ et $\langle 6,3,1,4\rangle$ sont également synonymes.

Certaines tâches sont "incluses" dans d'autres tâches (par exemple, tous les vecteurs noyaux de chaque tâche sont inclus dans l'ensemble noyau de la tâche $\langle 6,3,0,6\rangle$-GSB, mais certaines tâches ne sont pas comparables du point de vue de l'inclusion (par exemple les tâches $\langle 6,3,1,4\rangle$-GSB et $\langle 6,3,0,3\rangle$-GSB).

vecteur noyau $\rightarrow$ tâche $\downarrow$	canonique	$[6,0,0]$	$[5,1,0]$	$[4,2,0]$	$[4,1,1]$	$[3,3,0]$	$[3,2,1]$	$[2,2,2]$
$\langle 6,3,0,6\rangle$	oui	x	x	x	x	x	x	x
$\langle 6,3,1,6\rangle$					x		x	x
$\langle 6,3,0,5\rangle$	oui		x	x	x	x	x	x
$\langle 6,3,1,5\rangle$					x		x	x
$\langle 6,3,2,5\rangle$								x
$\langle 6,3,0,4\rangle$	oui			x	x	x	x	x
$\langle 6,3,1,4\rangle$	oui				x		x	x
$\langle 6,3,2,4\rangle$								x
$\langle 6,3,0,3\rangle$	oui					x	x	x
$\langle 6,3,1,3\rangle$	oui						x	x
$\langle 6,3,2,3\rangle$								x
$\langle 6,3,0,2\rangle$								x
$\langle 6,3,1,2\rangle$								x
$\langle 6,3,2,2\rangle$	oui							x

TABLE 6.1 – Noyaux de tâches $\langle n,m,\ell,u\rangle$-GSB (avec $n=6$ et $m=3$)

Remarque Il est important de remarquer que si un ensemble de vecteurs peut être associé à une tâche, tous les ensembles de vecteurs ne définissent pas une tâche GSB. Par exemple, on peut voir dans la table 6.1 que l'ensemble de vecteurs $\{[5,1,0],[4,2,0]\}$ ne définit pas une tâche GSB.

6.2.2.2 Les classes de tâches ℓ-ancrées, u-ancrées et (ℓ,u)-ancrées

Cette section présente des sous-classes des tâches GSB qui nous donne un meilleur aperçu de leur structure. Plus précisément, en observant les tâches décrites dans la table 6.1, on peut remarquer que plusieurs d'entre elles sont synonymes. Il est donc important d'avoir un seul représentant pour toutes les tâches $\langle n,m,\ell,u\rangle$-GSB qui sont en fait une seule et même tâche. Ceci est capturé par les notions de tâches ℓ-ancrées et u-ancrées.

Définition 6.14 (Ancrage)**.** *Soient G une tâche $\langle n,m,\ell,u\rangle$-GSB, G' la tâche $\langle n,m,\ell,\min(n,u+1)\rangle$-GSB et G'' la tâche $\langle n,m,\max(0,\ell-1),u\rangle$-GSB. G est* ℓ-ancrée *si G et G' sont synonymes. G est* u-ancrée *si G et G'' sont synonymes. G est* (ℓ,u)-ancrée *si elle est à la fois ℓ-ancrée et u-ancrée.*

Donc si G est ℓ-ancrée, augmenter la borne supérieure u ne modifie pas la tâche et, si G est u-ancrée, augmenter la borne inférieure ℓ ne modifie pas la tâche.

Comme exemple, considérons la famille des tâches $\langle 20,4,-,-\rangle$-GSB. On peut vérifier facilement que $\langle 20,4,4,8\rangle$ est une tâche ℓ-ancrée, $\langle 20,4,2,6\rangle$ est une tâche u-ancrée, $\langle 20,4,5,5\rangle$ est une tâche (ℓ,u)-ancrée tandis que $\langle 20,4,4,6\rangle$ n'est ni ℓ-ancrée ni u-ancrée.

Remarquons que toutes les tâches $\langle n,m,\ell,n\rangle$-GSB (resp. $\langle n,m,0,u\rangle$-GSB) sont ℓ-ancrées (resp. u-ancrées). Ces tâches sont dites *trivialement* ancrées.

Représentant canonique d'une tâche GSB Étant donnée une tâche $\langle n, m, \ell, u\rangle$-GSB ℓ-ancrée, son *représentant canonique* est la tâche $\langle n, m, \ell, u'\rangle$-GSB telle que la tâche $\langle n, m, \ell, u' - 1\rangle$-GSB n'est pas ℓ-ancrée. Une définition similaire s'applique aux tâches u-ancrées. Une tâche qui n'est ni u-ancrée ni ℓ-ancrée, ou qui est (ℓ, u)-ancrée, est son propre représentant.

Prenons comme exemple la table 6.1.

La tâche $\langle 6, 3, 2, 2\rangle$-GSB, qui est (ℓ, u)-ancrée, est la représentante des six tâches associées au seul vecteur noyau $[2, 2, 2]$. La tâche $\langle 6, 3, 1, 4\rangle$-GSB, qui est ℓ-ancrée, est la représentante des trois tâches associées à l'ensemble noyau $\{[4, 1, 1], [3, 2, 1], [2, 2, 2]\}$. Finalement, la tâche $\langle 6, 3, 1, 3\rangle$-GSB, qui n'est pas ancrée, est sa propre représentante : c'est la seule tâche associée à l'ensemble noyau $\{[3, 2, 1], [2, 2, 2]\}$.

Quand on considère la table 6.1, il y a sept représentants canoniques. Ces tâches canoniques sont représentées dans la figure 6.1 où "$A \to B$" signifie "A inclut strictement B". Remarquons que la tâche représentante $\langle 6, 3, 1, 3\rangle$-GSB n'est pas ancrée.

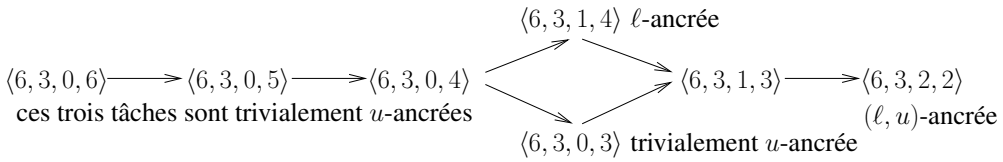

FIGURE 6.1 – Les tâches canoniques $\langle n, m, -, -\rangle$-GSB sont partiellement ordonnées

6.2.2.3 Une caractérisation des tâches GSB ℓ-ancrées et u-ancrées

Rappelons qu'une tâche est *faisable* si son ensemble de vecteurs de sortie $\mathcal{O}$ n'est pas vide.

Théorème 6.15. *Soit T une tâche $\langle n, m, \ell, u\rangle$-GSB faisable. T est ℓ-ancrée si et seulement si $u \geq n - \ell(m - 1)$.*

Démonstration. Supposons d'abord que $n - \ell(m-1) > u \geq \ell$. Comme $n - \ell(m-1) \geq u+1$, il existe un vecteur (avec m entrées) dont la première entrée est égale à $u + 1$ qui est un vecteur noyau de la tâche $\langle n, m, \ell, u + 1\rangle$-GSB. Mais comme $u + 1 > u$, ce vecteur ne peut pas être un vecteur noyau de la tâche $\langle n, m, \ell, u\rangle$-GSB. La tâche $\langle n, m, \ell, u\rangle$-GSB ne peut donc pas être ℓ-ancrée.

Supposons maintenant que $u \geq n - \ell(m-1) \geq \ell$ et considérons le vecteur quantité $[n - \ell(m - 1), \ell, \ldots, \ell]$ (avec m entrées). La somme de toutes ses entrées est n. Parce que l'entrée égale à $n-\ell(m-1)$ est la seule valeur supérieure à ℓ, c'est la plus grande valeur qui peut apparaitre dans un vecteur noyau des deux tâches $\langle n, m, \ell, u\rangle$-GSB et $\langle n, m, \ell, u + 1\rangle$-GSB pour tout $u \geq n - \ell(m - 1)$. Les tâches $\langle n, m, \ell, u\rangle$-GSB et $\langle n, m, \ell, u+1\rangle$-GSB sont donc la même tâche, d'où on conclut que $\langle n, m, \ell, u\rangle$-GSB est ℓ-ancrée. □

Théorème 6.16. *Soit T une tâche $\langle n, m, \ell, u\rangle$-GSB faisable. T est u-ancrée si et seulement si $\ell \leq n - u(m - 1)$.*

Démonstration. Le raisonnement est similaire à celui du théorème 6.15. □

Le corollaire suivant est une conséquence des deux théorèmes précédents.

Corollaire 6.17. *Soit $\ell \leq \frac{n}{m} \leq u$. La tâche $\langle n, m, \ell, \max(\ell, n - \ell(m - 1))\rangle$-GSB est ℓ-ancrée, tandis que la tâche $\langle n, m, \max(0, n - u(m - 1)), u\rangle$-GSB est u-ancrée.*

6.2.2.4 Les résultats structurels

Lemme 6.18. *Soit T une tâche $\langle n, m, \ell, u\rangle$-GSB quelconque. Soit $u' \geq u$ et T' la tâche $\langle n, m, \ell, u'\rangle$-GSB. Nous avons alors $S(T) \subseteq S(T')$.*

Démonstration. La seule différence entre T et T' est la borne supérieure sur le nombre de processus qui peuvent décider chaque valeur. Si au plus u processus décident chaque valeur, alors nécessairement moins de u' processus décident chaque valeur, et donc chaque vecteur de sortie de la tâche $\langle n, m, \ell, u\rangle$-GSB T est aussi un vecteur de sortie de la tâche $\langle n, m, \ell, u'\rangle$-GSB T' et donc $S(T) \subseteq S(T')$. □

Lemme 6.19. *Soit T une tâche $\langle n, m, \ell, u\rangle$-GSB quelconque. Soit $\ell' \leq \ell$ et T' la tâche $\langle n, m, \ell', u\rangle$-GSB. Nous avons alors $S(T) \subseteq S(T')$.*

Démonstration. Le raisonnement est similaire à celui du lemme 6.18. □

Le prochain théorème caractérise la tâche la plus difficile de la sous-famille de tâches $\langle n, m, -, -\rangle$-GSB. Rappelons que T_1 est plus difficile que T_2 si $S(T_1) \subset S(T_2)$.

Théorème 6.20. *La tâche $\langle n, m, \lfloor \frac{n}{m} \rfloor, \lceil \frac{n}{m} \rceil\rangle$-GSB est la tâche la plus difficile de la famille des tâches $\langle n, m, -, -\rangle$-GSB faisables.*

Démonstration. Comme nous ne considérons que les tâches faisables, nous avons $\ell \leq \frac{n}{m} \leq u$. La démonstration est alors une conséquence directe des lemmes 6.18 et 6.19. □

Observons que, étant donnés n and m, la tâche $\langle n, m, \lfloor \frac{n}{m} \rfloor, \lceil \frac{n}{m} \rceil\rangle$-GSB n'est pas nécessairement ancrée. Par exemple, la tâche $\langle 10, 4, 2, 3\rangle$-GSB n'est ni ℓ-ancrée ni u-ancrée tandis que la tâche $\langle 10, 5, 2, 2\rangle$-GSB est (ℓ, u)-ancrée.

Théorème 6.21. *Soient T une tâche $\langle n, m, \ell, u\rangle$-GSB faisable, $T1$ la tâche $\langle n, m, \ell', u\rangle$-GSB où $\ell' = n - u(m-1)$ et $T2$ la tâche $\langle n, m, \ell, u'\rangle$-GSB où $u' = n - \ell(m-1)$. Nous avons les propriétés suivantes : (i) $(\ell' \geq \ell) \Rightarrow S(T1) \subseteq S(T)$ et (ii) $(u' \leq u) \Rightarrow S(T2) \subseteq S(T)$.*

Démonstration. Nous démontrons le théorème pour le cas (i) (la démonstration du cas (ii) est similaire). Montrons d'abord que la tâche $\langle n, m, \ell', u\rangle$-GSB est faisable, c'est-à-dire que $\ell' \leq \frac{n}{m} \leq u$. Observons que, comme la tâche $\langle n, m, \ell, u\rangle$-GSB est faisable, nous avons $\frac{n}{m} \leq u$. Nous devons donc montrer que $\ell' \leq \frac{n}{m}$, ce qui est obtenu de la façon suivante (rappelons que $m > 1$) :

$$\begin{array}{rlrl} & n/m \leq \ u & \Leftrightarrow & n \leq \ u \cdot m \\ \Leftrightarrow & n(m-1) \leq \ u \cdot m(m-1) & \Leftrightarrow & n \cdot m - u \cdot m^2 + u \cdot m \leq \ n \\ \Leftrightarrow & \ell' = n - u \cdot m + u \leq \ n/m. & & \end{array}$$

Comme $\ell' = n - u(m-1) \leq \frac{n}{m} \leq u$, le vecteur de taille m $[u, \ldots, u, \ell']$ est un vecteur noyau de la tâche $\langle n, m, \ell', u\rangle$-GSB faisable. Comme $\ell' \geq \ell$, ce vecteur est aussi un vecteur noyau de la tâche $\langle n, m, \ell, u\rangle$ GSB, ce qui conclut la démonstration pour le cas (i). □

Le théorème suivant identifie le représentant canonique de toute tâche $\langle n, m, \ell, u\rangle$-GSB faisable.

Théorème 6.22. *Soient T une tâche $\langle n, m, \ell, u\rangle$-GSB faisable et $f()$ la fonction $f(\ell, u) = (\ell', u')$ où $\ell' = \max(\ell, n - u(m-1))$ et $u' = \min(u, n - \ell(m-1))$. Le représentant canonique de T est la tâche $\langle n, m, \ell_{fp}, u_{fp}\rangle$-GSB T_{fp} où la paire (ℓ_{fp}, u_{fp}) est le point fixe de $f(\ell, u)$.*

Démonstration. Observons d'abord que, en utilisant le même raisonnement que pour le théorème6.21, nous avons $\ell' \leq \frac{n}{m} \leq u'$, et donc T_{fp} est faisable (lemme 6.9). De plus, selon la définition de ℓ' et u', nous avons aussi que $0 \leq \ell \leq \ell' \leq \frac{n}{m} \leq u' \leq u \leq n$. Nous considérons quatre cas.

- Cas $\ell \geq n - u(m-1)$ et $u \leq n - \ell(m-1)$. Nous avons alors trivialement $\ell' = \ell$ et $u' = u$, d'où on conclut que $S(T)$ et $S(T_{fp})$ ont les mêmes vecteurs noyaux.
- Cas $\ell' = n - u(m-1) > \ell$ et $u' = u$. Considérons le vecteur noyau de T qui a autant d'entrées que possible égales à $u = u'$. Cela signifie que ce vecteur a $m-1$ entrées égales à $u = u'$ et que sa dernière entrée est égale à $n - u'(m-1)$, c'est-à-dire égale à ℓ'. $S(T)$ n'a donc aucun vecteur noyau avec une entrée égale à $\ell'' < \ell'$. On conclut de cette observation que les vecteurs noyaux de T sont aussi les vecteurs noyaux de T_{fp}, c'est-à-dire que $S(T) = S(T_{fp})$.
- Cas $\ell' = \ell$ et $u' = n - \ell(m-1) < u$. Ce cas est similaire au cas précédent. Considérons le vecteur noyau de T qui a autant d'entrées que possible égales à $\ell = \ell'$. Cela signifie que ce vecteur a $m-1$ entrées égales à $\ell = \ell'$ et que sa dernière entrée est égale à $n - \ell'(m-1)$, c'est-à-dire égale à u'. $S(T)$ n'a donc aucun vecteur noyau avec une entrée égale à $u'' > u'$. Les vecteurs noyaux de T sont donc aussi les vecteurs noyaux de T_{fp}, c'est-à-dire que $S(T) = S(T_{fp})$.
- Cas $\ell' = n - u(m-1) > \ell$ et $u' = n - \ell(m-1) < u$. Ce cas est une simple combinaison des cas précédents (un cas traite les vecteurs noyaux de T avec les entrées aussi grandes que possibles, tandis que l'autre cas traite les vecteurs noyaux de T avec les entrées aussi petites que possibles).

Selon les théorèmes 6.15 et 6.16, ni la tâche $\langle n, m, \ell'', u\rangle$-GSB avec $\ell'' > \ell'$ ni la tâche $\langle n, m, \ell, u''\rangle$-GSB avec $u'' < u'$ ne sont des synonymes de T, ce qui conclut la démonstration du théorème. □

6.2.3 Calculabilité

Rappelons que pour une tâche $\langle n, m, \vec{\ell}, \vec{u}\rangle$-GSB $T = (\mathcal{I}, \mathcal{O}, \Delta)$, nous avons $\Delta(I) = \Delta(I') = \mathcal{O}$ pour toute paire de vecteurs d'entrée I, I'. À première vue, on pourrait alors penser qu'une solution triviale pour T serait de choisir un vecteur de sortie $O \in \mathcal{O}$ prédéfini et de toujours décider sans aucune communication, quel que soit le vecteur d'entrée. Ce n'est en fait pas possible à cause de la propriété d'indépendance de l'index : le comportement d'un processus doit être indépendant de son index. En fait, certaines tâches GSB (faisables) ne peuvent pas être résolues dans le modèle doté uniquement de registres.

6.2.3.1 Universalité de la tâche $\langle n, n, 1, 1\rangle$-GSB

Quand on considère la famille des tâches GSB, on peut se poser la question suivante : existe-t-il une tâche GSB universelle ? En d'autres termes, existe-t-il une tâche GSB qui permet de résoudre toutes les autres tâches GSB sur n processus ? La réponse est "oui". Le théorème suivant montre que la tâche $\langle n, n, 1, 1\rangle$-GSB (renommage parfait) permet de résoudre n'importe quelle tâche GSB sur n processus. Le renommage parfait est donc *universel* pour la famille des tâches $\langle n, -, -, -\rangle$-GSB.

Comme nous le verrons avec le corollaire 6.25, la tâche $\langle n, n, 1, 1\rangle$-GSB (renommage parfait) ne peut pas être résolue quand les processus n'ont accès qu'à des registres.

Théorème 6.23. *Toute tâche $\langle n, m, \vec{\ell}, \vec{u}\rangle$-GSB (faisable) peut être résolue à partir d'une solution à la tâche $\langle n, n, 1, 1\rangle$-GSB.*

Démonstration. Observons d'abord que la tâche $\langle n, n, 1, 1\rangle$-GSB a un seul vecteur noyau, à savoir $[1, \ldots, 1]$. Étant donné un algorithme qui résout cette tâche, soit dec_i sa sortie au processus p_i.

Pour résoudre la tâche symétrique $\langle n, m, \ell, u\rangle$-GSB, les processus exécutent un algorithme qui résout la tâche $\langle n, n, 1, 1\rangle$-GSB et un processus p_i prend $output_i = ((dec_i - 1) \mod m) + 1$ comme sortie. Le vecteur noyau correspondant est alors $[\lceil \frac{m}{n} \rceil, \ldots, \lceil \frac{m}{n} \rceil, \lfloor \frac{m}{n} \rfloor, \ldots, \lfloor \frac{m}{n} \rfloor]$. Selon l'hypothèse de faisabilité, nous avons $\ell \leq \frac{m}{n} \leq u$. Comme ℓ et u sont des entiers, nous avons $\ell \leq \lfloor \frac{m}{n} \rfloor \leq \lceil \frac{m}{n} \rceil \leq u$. Le vecteur $[\lceil \frac{m}{n} \rceil, \ldots, \lceil \frac{m}{n} \rceil, \lfloor \frac{m}{n} \rfloor, \ldots, \lfloor \frac{m}{n} \rfloor]$ est donc un vecteur noyau de la tâche $\langle n, m, \ell, u\rangle$-GSB.

Pour résoudre la tâche asymétrique $\langle n, m, \vec{\ell}, \vec{u}\rangle$-GSB, nous considérons d'abord l'ensemble de vecteurs de sortie $\mathcal{O}$. Nous ordonnons alors ces vecteurs de la même manière déterministe et choisissons le premier d'entre eux. Soit V ce vecteur de sortie de la tâche $\langle n, m, \vec{\ell}, \vec{u}\rangle$-GSB. Nous utilisons alors le

même vecteur V pour tous les processus. Soit dec_i la valeur obtenue par p_i dans la tâche $\langle n, n, 1, 1\rangle$-GSB. Un processus p_i prend alors l'entrée $V[dec_i]$ comme sortie $output_i$ pour la tâche $\langle n, m, \vec{\ell}, \vec{u}\rangle$-GSB. Parce que la tâche $\langle n, n, 1, 1\rangle$-GSB a un seul vecteur noyau $[1, \ldots, 1]$, chaque entrée de V est choisie par un seul processus. Cela satisfait la spécification de la tâche $\langle n, m, \vec{\ell}, \vec{u}\rangle$-GSB, ce qui conclut la démonstration du théorème. □

Théorème 6.24. *Pour tout n, la tâche $(n-1)$-accord ensembliste peut être résolue de manière sans-attente en utilisant des registres et une solution à la tâche $\langle n, n, 1, 1\rangle$-GSB.*

Démonstration. Considérons un algorithme A qui résout la tâche $\langle n, n, 1, 1\rangle$-GSB. Selon le théorème 6.10, nous pouvons supposer que A est basé sur des comparaisons. Soit E une exécution de A dans laquelle seul le processus p_i participe. Parce que A est basé sur des comparaisons et est indépendant de l'index de p_i, la sortie de p_i dans E n'est pas une fonction de son entrée. Dans chaque exécution dans laquelle seul p_i participe, p_i obtiendra donc la même sortie λ. Comme p_i obtient λ dans chaque exécution dans laquelle il participe seul et A est indépendant de l'index, dans chaque exécution dans laquelle seul p_j participe, $j \neq i$, p_j obtiendra également la même sortie λ.

Soit $\overline{\lambda} \in \{1, \ldots, n\}$ une valeur différente de λ. En utilisant A et $\overline{\lambda}$ le $(n-1)$-accord ensembliste peut être résolu de la façon suivante (l'indépendance de l'index n'est pas requise pour le $(n-1)$-accord ensembliste). Chaque processus p_i annonce d'abord sa proposition v_i en écrivant v_i dans $M[i]$ (M est un tableau registres partagés initialisés à $\perp$) puis exécute A en utilisant son index comme entrée.

- Si p_i obtient une valeur différente de $\overline{\lambda}$, il décide sa proposition v_i.
- Si p_i obtient $\overline{\lambda}$, cela signifie qu'un autre processus p_j a commencé à exécuter A après avoir écrit sa proposition v_j dans $M[j]$. Dans ce cas, p_i décide n'importe quelle entrée $M[j]$ telle que $j \neq i$ et $M[j] \neq \perp$. Le processus p_i est alors le seul à ne pas décider sa propre proposition.

Cette implémentation satisfait de façon triviale la propriété de validité de l'accord ensembliste et la terminaison sans-attente. Si un processus obtient $\overline{\lambda}$ dans A, au moins deux processus participent et ont écrit leurs valeurs dans M (A produit la sortie $\lambda \neq \overline{\lambda}$ si un processus l'exécute seul) et deux processus décident la même valeur, ce qui implique la propriété d'accord et conclut la démonstration du théorème. □

Le corollaire suivant est la conséquence du théorème 6.24 et du fait que pour tout n, le $(n-1)$-accord ensembliste ne peut pas être résolu de manière sans-attente en utilisant uniquement des registres [27, 75, 112].

Corollaire 6.25. *Le renommage parfait ne peut pas être résolu de manière sans-attente en utilisant uniquement des registres.*

Théorème 6.26. *Pour tout $n \geq 3$, la tâche $\langle n, n, 1, 1\rangle$-GSB ne peut pas être résolue de manière sans-attente en utilisant des registres et une solution à la tâche $(n-1)$-accord ensembliste.*

Démonstration. Supposons par contradiction qu'il existe un algorithme A qui résolve la tâche $\langle n, n, 1, 1\rangle$-GSB en utilisant uniquement une solution au $(n-1)$-accord ensembliste et des registres. Considérons alors l'ensemble des exécutions S de l'algorithme A dans lesquels seuls les $n-1$ processus $p_1, \ldots, p_{n-1}$ participent. La spécification du $(n-1)$-accord ensembliste implique que S contient un sous-ensemble S' d'exécutions dans lesquelles chaque invocation par p_i, $1 \leq i \leq n-1$, de la solution au $(n-1)$-accord ensembliste lui renvoie la valeur qu'il a proposé (parce que seuls $n-1$ processus participent).

Nous pouvons alors obtenir un algorithme B pour $n-1$ processus à partir de A en remplaçant chaque invocation du $(n-1)$-accord ensembliste par l'identité (la sortie est égale à l'entrée). Pour chaque exécution E de B, il existe une exécution dans S' dans laquelle les processus ont le même comportement et donc les mêmes sorties. L'algorithme B résout donc la tâche $\langle n-1, n, 0, 1\rangle$-GSB en utilisant uniquement des registres. Mais il a été prouvé dans [17] que pour tout $n \geq 3$, la tâche $\langle n-1, n, 0, 1\rangle$-GSB ($n$-renommage pour $n-1$ processus) ne peut pas être résolue en utilisant uniquement des registres, une contradiction qui conclut la démonstration du théorème. □

6.2.4 De la tâche "slot" au renommage

Cette section présente un algorithme simple qui résout la tâche du $(n+1)$-renommage (c'est-à-dire la tâche $\langle n, n+1, 0, 1\rangle$-GSB) de manière sans attente en utilisant des registres et une solution à la tâche du $(n-1)$-slot (c'est-à-dire la tâche $\langle n, n-1, 1, 2\rangle$-GSB). L'objet qui résout la tâche $\langle n, n-1, 1, 2\rangle$-GSB utilisé dans l'algorithme est noté KS. Il fournit aux processus une seule opération notée $\mathsf{slot\_request}_{n-1}()$. Dans cette opération, dont la sémantique a été décrite dans la section 6.2.1.1, chaque valeur x, $1 \leq x \leq n-1$, est décidée par au moins un processus (quand tous participent et sont corrects).

Objets partagés utilisés En plus de KS, les processus coopèrent en utilisant un objet snapshot $STATE[1..n]$. Chaque entrée $STATE[i]$ est initialisée à $\perp$ et n'est modifiée que par le processus correspondant p_i. Le processus p_i y écrit une paire d'entiers $\langle my\_slot_i, id_i\rangle$ (où my_slot_i est la valeur qu'il obtient de KS et id_i son identité).

```
operation new_name() :
(01) my_slot_i ← KS.slot_request_{n-1}() ;
(02) STATE[i] ← ⟨my_slot_i, id_i⟩ ; (slot_i[1..n], ids_i[1..n]) ← STATE.snapshot() ;
(03) if (∀j ≠ i : slot_i[j] ≠ my_slot_i)
(04)    then return(my_slot_i)
(05)    else let j ≠ i such that slot_i[j] = my_slot_i ;
(06)         if (id_i < ids_i[j]) then return(n) else return(n + 1) end if
(07) end if.
```

FIGURE 6.2 – Solution au $(n+1)$-renommage en utilisant une solution au $(n-1)$-slot (tâche $\langle n, n-1, 1, 2\rangle$-GSB) (code de p_i)

Comportement des processus Chaque processus p_i gère deux tableaux locaux $slot_i[1..n]$ et $ids_i[1..n]$. Ces tableaux sont utilisés pour stocker les valeurs lues dans les deux champs des entrées de l'objet snapshot $STATE$. L'algorithme pour le processus p_i est présenté dans la figure 6.2. Il est constitué de deux parties.

- Un processus p_i commence par acquérir un numéro de slot (ligne 01). Il écrit ensuite ses attributs (numéro de slot et identité) dans $STATE[i]$, puis lit l'objet snpashot pour avoir une vue globale "atomique" de tous les attributs qui ont été écrits (ligne 02).
- Le processus p_i détermine alors son nouvel identifiant. Si il n'observe aucun autre processus avec le même numéro de slot, ce numéro de slot devient son nouvel identifiant. (lignes 03-04). Dans le cas contraire, les propriétés de l'objet KS impliquent qu'il y a exactement un autre processus p_j qui a le même numéro de slot (ligne 05). Les processus p_i et p_j sont alors en compétition pour un nouvel identifiant. De plus, il est possible que p_j considère déjà son numéro de slot comme son nouvel identifiant. Le processus p_i résout ce conflit selon l'ordre entre son identité et celle de p_j : si l'identité de p_i est plus petite, p_i décide n, sinon il décide $n+1$ (ligne 06).

Théorème 6.27. *L'algorithme décrit dans la figure 6.2 résout la tâche du $(n+1)$-renommage de manière sans-attente à partir de n'importe quelle solution à la tâche $(n-1)$-slot.*

Démonstration. La terminaison sans attente et le fait que les nouveaux identifiants appartiennent à l'ensemble $\{1, \ldots, n+1\}$ sont des conséquences directes du texte de l'algorithme. Nous nous concentrons donc sur la démonstration du fait que deux processus ne peuvent pas obtenir le même nouvel identifiant.

À cause des propriétés de l'objet KS, $n-2$ obtiennent des slots uniques parmi $[1..n-1]$. Soient p_x et p_y les processus qui reçoivent le même slot s. La démonstration se base sur le fait que les invocations $\mathsf{snapshot}()$ sont ordonnées suivant un ordre total. Il y a deux cas.

– La valeur du snapshot obtenu par p_x est telle que $STATE[y] = \perp$. Dans ce cas, p_x renvoie s comme nouvel identifiant. La valeur du snapshot obtenu par p_y contiendra alors $STATE[x] = s$. p_y obtiendra alors le nouvel identifiant n ou $n + 1$ selon les valeurs de id_x et id_y.
– Les valeurs des snapshots obtenus par p_x et p_y sont toutes les deux telles que les deux entrées $STATE[x]$ et $STATE[y]$ sont égales à s. Dans ce cas, les deux processus exécutent les lignes 05-06, ce qui implique qu'ils obtiennent les nouveaux identifiants n et $n + 1$ en fonction de l'ordre entre id_x et id_y.

□

Vers un algorithme général Les tâches $(2n - 2)$-renommage ($\langle n, 2n - 2, 0, 1\rangle$-GSB) et cassage de symétrie faible (Weak Symmetry Breaking, $\langle n, 2, 1, n - 1\rangle$-GSB) sont équivalentes (voir par exemple [35]). Comme les tâches de cassage de symétrie faible et 2-slot sont en fait une seule et même tâche, les tâches $(2n - 2)$-renommage et 2-slot sont équivalentes.

Plus généralement, quand on considère le problème plus général de trouver un algorithme qui résout le $(2n - k)$-renommage en utilisant le k-slot, l'algorithme présenté dans la figure 6.2 est une réponse spécifique au cas $k = n - 1$, tandis que l'équivalence entre $(2n - 2)$-renommage et cassage de symétrie faible est une réponse spécifique au cas $k = 2$.

La question "existe-t-il un algorithme général qui résout le $(2n - k)$-renommage en utilisant le k-slot, et plus généralement les tâches $(2n - k)$-renommage et k-slot sont-elles équivalentes ?" reste donc ouverte.

Chapitre 7

Conclusion

La synchronisation de processus dans un système distribué est un sujet vaste et difficile. Quand les processus qui constituent le système sont asynchrones et que certains d'entre eux peuvent subir des défaillances, certains problèmes deviennent impossibles à résoudre. Ils peuvent néanmoins devenir possibles si on affaiblit la condition de progression requise, c'est-à-dire les conditions dans lesquelles un processus correct doit terminer une opération qu'il exécute.

7.1 Contributions

Dans ce document, nous nous sommes intéressés au lien entre calculabilité des objets distribués et conditions de progression dans un système constitué de processus asynchrones qui sont susceptibles de subir des pannes. Les contributions suivantes ont été apportées.

Conditions de progression asymétriques Les conditions de progression classiques imposent les mêmes conditions à tous les processus sans distinction. Le chapitre 3 a introduit la notion de conditions de progression *asymétriques* : des conditions de progression qui peuvent imposer différentes conditions de terminaison à différents processus. Il a ensuite introduit la condition de progression (n, x)-*vivacité* et montre qu'elle définit une nouvelle hiérarchie stricte d'objets. Finalement, il a introduit la condition de progression x-*sans-attente* qui permet de résoudre le problème du consensus (tous les processus doivent s'accorder sur une seule valeur proposée) dans un système dans lequel seul un sous-ensemble de processus doivent toujours décider une valeur.

Abstractions Le chapitre 4 a considéré des abstractions : des objets qui sont utilisés pour faciliter la conception d'algorithmes mais qui ne changent pas la calculabilité des objets dans le modèle. Il a d'abord introduit un modèle de système hybride dans lequel l'ensemble des processus est partitionné, chaque partition ayant accès à une mémoire partagée qui n'est pas accessible aux processus qui ne font pas partie de cette partition. Les processus de partitions différentes peuvent communiquer en envoyant et en recevant des messages. Il a ensuite montré que, dans ce modèle, une mémoire partagée globale ne peut pas être implémentée de manière sans-attente sans aide additionnelle et précise l'aide nécessaire et suffisante pour y parvenir.

L'abstraction du *snapshot* permet d'écrire dans un registre et de lire de manière atomique les valeurs de tous les registres du système (prendre un snapshot). Quand on considère le modèle dans lequel les processus ont accès à une mémoire partagée globale, l'abstraction du snapshot ne renforce pas le système : elle peut être implémentée de manière sans-attente sans aide additionnelle. Durant une écriture, les algorithmes de snapshot classiques calculent d'abord l'ensemble des valeurs des registres du système, puis écrivent. Un nouvel algorithme a été présenté qui écrit, puis calcule les valeurs des autres registres.

De plus, cet algorithme permet de prendre un snapshot partiel (c'est-à-dire de seulement un sous-ensemble des registres du système) de manière efficace.

Simulations Une simulation permet de comparer différents modèles de systèmes en déterminant si un modèle peut être implémenté dans un autre modèle. Un travail à la base de ce domaine est la simulation *Borowsky-Gafni*. La simulation BG montre que, quand on considère un modèle dans lequel les processus n'ont accès qu'à des registres partagés, un objet peut être implémenté de manière t-résiliente (les processus doivent décider une valeur si au plus t d'entre eux crashent) si et seulement si l'objet peut être implémenté de manière sans-attente dans un système de $t + 1$ processus. La simulation originale ne considère que des objets pour lesquels un processus peut décider n'importe quelle valeur déjà décidée par un autre processus ; elle a ensuite été étendue au cas général. Le chapitre 5 a d'abord présenté une manière alternative de réaliser la version étendue de la simulation BG. Il a ensuite montré, en utilisant des simulations, qu'un objet peut être implémenté dans un système où les processus ont accès au x-consensus (tous les sous-ensembles d'au plus x processus peuvent résoudre le consensus) et où au plus t processus peuvent crasher si et seulement si le même objet peut être implémenté dans un système où les processus n'ont accès qu'à des registres partagés et où au plus $\lfloor t/x \rfloor$ processus peuvent crasher.

Cassage de la symétrie Le chapitre 6 a finalement considéré les tâches de *cassage de la symétrie* : des tâches dans lesquelles des processus qui démarrent dans le même état, à part un identifiant unique mais non borné, doivent décider une valeur bornée. Les conditions sur cette valeur dépendent de la tâche, la condition minimale étant que tous les processus ne décident pas la même valeur. Une tâche de cassage de la symétrie bien connue est le M-renommage, dans laquelle tous les processus doivent décider une valeur différente prise dans $[1..M]$. Dans la version *adaptative* du renommage, le nombre p de processus participants n'est pas connu et M est une fonction de p. La condition de progression x-sans-obstruction requiert qu'un processus décide une valeur en un temps fini si, après un instant t, au plus x processus exécutent leur programme. Un algorithme qui résout le $(p + x - 1)$-renommage adaptatif de manière x-sans-obstruction a été présenté.

Finalement, le concept de tâches de *cassage de la symétrie généralisé* (tâches GSB, *Generalized Symmetry Breaking tasks*) a été présenté. Ce concept unifie dans un seul cadre de nombreux problèmes qui étaient considérés auparavant comme indépendants, tels que le renommage non-adaptatif et le cassage de symétrie faible (*Weak Symmetry Breaking*). Une tâche GSB est définie par quatre paramètres : le nombre n de processus qui peuvent accéder à la tâche, le nombre m de valeurs de sortie possibles et les bornes minimale ℓ et maximale u sur le nombre de processus qui peuvent décider chaque valeur. Par exemple, la tâche GSB $\langle n, n, 1, 1\rangle$ correspond au renommage parfait dans lequel la taille de l'espace des noms correspond exactement au nombre de processus dans le système. Pour chaque n, la famille de tâches GSB $\langle n, -, -, -\rangle$ peuvent être ordonnées partiellement selon leurs vecteurs de sortie. Après la présentation de différents résultats caractérisant la calculabilité et la puissance relative des tâches GSB, il a été montré que la tâche GSB $\langle n, n, 1, 1\rangle$ (renommage parfait) est strictement plus fort que le $(n - 1)$-accord ensembliste dans lequel n processus doivent s'accorder sur au plus $n - 1$ valeurs proposées).

7.2 Perspectives

Dans le chapitre 3, nous n'avons étudié que le problème du consensus. Il serait intéressant d'étudier l'impact des conditions de progression asymétriques sur d'autres tâches, en particulier sur les tâches colorées.

La section 5.2 ne concerne que les tâches non colorées. Dans [86], nous donnons une extension aux tâches colorées, mais elle ne fonctionne que dans certains cas. Une extension à l'ensemble des cas serait souhaitable.

Castañeda et Rajsbaum ont montré dans [32] que la tâche $\langle n, 2, 1, n-1\rangle$-GSB (Weak Symmetry Breaking) peut être résolue de manière sans-attente pour certaines valeurs de n, mais n'ont pas donné d'algorithme permettant de le faire. Il serait intéressant de découvrir un tel algorithme ; cela permettrait de résoudre le $(2n-2)$-renommage non adaptatif (tâche $\langle n, 2n-2, 0, 1\rangle$-GSB) [61] pour ces valeurs particulières de n.

Les tâches GSB définies dans ce document ne sont pas adaptatives ; les sorties des processus dépend du nombre total de processus dans le système, pas du nombre de processus qui invoquent l'opération. Une définition similaire, mais pour les tâches adaptatives, permettrait d'englober des tâches telles que le renommage adaptatif ou la version de décision du Test&Set.

Bibliographie

[1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic Snapshots of Shared Memory. *Journal of the ACM*, 40(4) :873-890, 1993.

[2] Afek Y., Gafni E. and Lieber., Tight Group Renaming on Groups of Size g Is Equivalent to g-Consensus. *Proc. 23rd Int'l Symposium on Distributed Computing (DISC'09)*, Springer Verlag LNCS #5805, pp. 111-126, 2009.

[3] Afek Y., Gafni E. and Morisson A., Common2 Extended to Stacks and Unbounded Concurrency. *Proc. 25th ACM Symposium on Principles of Distributed Computing*, ACM Press, pp. 218-227, 2006.

[4] Afek Y., Gafni E., Rajsbaum S., Raynal M. and Travers C., Simultaneous consensus tasks : a tighter characterization of set consensus. *Proc. 8th Int'l Conference on Distributed Computing and Networking (ICDCN'06)*, Springer Verlag LNCS #4308, pp. 331-341, 2006.

[5] Afek Y., Gafni E., Rajsbaum S., Raynal M. and Travers C., The k-Simultaneous Consensus Problem. *Distributed Computing*, To appear, 2010, DOI 10.1007/s00446-009-0090-8.

[6] Afek Y., Gamzu I., Levy I., Merritt M., and Taubenfeld G., Group Renaming. *Proc. 12th International Conference on Principles of Distributed Systems (OPODIS'08)*, Springer Verlag LNCS #5401, pp. 58-72, 2008.

[7] Afek Y. and Merritt M., Fast, Wait-Free $(2k-1)$-Renaming. *Proc. 18th ACM Symposium on Principles of Distributed Computing (PODC'99)*, ACM Press, pp. 105-112, 1999.

[8] Afek Y., Stupp G., Touitou D., Long-lived Adaptive Collect with Applications. *Proc. 40th IEEE Symposium on Foundations of Computer Science Computing (FOCS'99)*, IEEE Computer Press, pp. 262-272, 1999.

[9] Afek Y., Weisberger E. and Weisman H., A Completeness Theorem for a Class of Synchronization Objects. *Proc. 12th ACM Symposium on Principles of Distributed Computing*, ACM Press, pp. 159-170, 1993.

[10] Aguilera M.K., A Pleasant Stroll Through the Land of Infinitely Many Creatures. *ACM Sigact News, Distributed Computing Column*, 35(2) :36-59, 2004.

[11] Anderson J., Composite Registers. *Proc. 9th ACM Symposium on Principles of Distributed Computing (PODC'90)*, pp. 15-29, 1990.

[12] Anderson J., Multi-writer Composite Registers. *Distributed Computing*, 7(4) :175-195, 1994.

[13] Angluin D., Local and Global Properties in Networks of Processors. *Proc. 12th ACM Symposium on Theory of Computing (STOC'80)*, ACM Press, pp. 82-93, 1980.

[14] Attiya H., Efficient and Robust Sharing of Memory in Message-passing Systems. *Journal of Algorithms*, 34(1) :109-127, 2000.

[15] Attiya H., Needed : Foundations for Transactional Memory. *ACM SIGACT News, Distributed Computing Column*, 39(1) :59-61, 2008.

[16] Attiya H., Bar-Noy A. and Dolev D., Sharing Memory Robustly in Message Passing Systems. *Journal of the ACM*, 42(1) :121-132, 1995.

[17] Attiya H., Bar-Noy A., Dolev D., Peleg D. and Reischuk R., Renaming in an Asynchronous Environment. *Journal of the ACM*, 37(3) :524-548, 1990.

[18] Attiya H. and Fouren A., Polynomial and Adaptive Long-lived $(2p-1)$-Renaming. *Proc. 14th Int'l Symposium on Distributed Computing (DISC'00)*, Springer Verlag LNCS #1914 , pp.149-163, 2000.

[19] Attiya H. and Fouren A., Algorithms Adapting to Contention Point. *Journal of the ACM*, 50(4) :444-468, 2003.

[20] Attiya H., Fouren A. and Gafni E., An Adaptive Collect Algorithm with Applications. *Distributed Computing*, 15 :87-96, 2002.

[21] Attiya H., Gorbach A. and Moran S., Computing in Totally Anonymous Asynchronous Shared Memory Systems. *Information and Computation*, 173(2) :162–183, 2002.

[22] Attiya H., Guerraoui R. and Ruppert E., Partial Snapshot Objects. *Proc. 20th ACM Symposium on Parallel Architectures and Algorithms (SPAA'08)*, ACM Press, pp. 336-343, 2008.

[23] Attiya H. and Rachman O., Atomic Snapshot in $O(n \log n)$ Operations. *SIAM Journal of Computing*, 27(2) :319-340, 1998.

[24] Attiya H. and Rajsbaum S., The Combinatorial Structure of Wait-Free Solvable Tasks, *SIAM Journal of Computing,* 31(4) :1286-1313, 2002.

[25] Attiya H. and Welch J., Distributed Computing : Fundamentals, Simulations and Advanced Topics, (2d Edition), *Wiley-Interscience*, 414 pages, 2004.

[26] Bonnet F. and Raynal M., A Simple Proof of the Necessity of the Failure Detector Σ to Implement an Atomic Register in Asynchronous Message-passing Systems. *Information Processing Letters*, 110(4) : 153-157, 2010.

[27] Borowsky E. and Gafni E., Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. *Proc. 25th ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 91-100, 1993.

[28] Borowsky E. and Gafni E., Immediate Atomic Snapshots and Fast Renaming. *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, pp. 41-51, 1993.

[29] Borowsky E. and Gafni E., The Implication of the Borowsky-Gafni Simulation on the Set Consensus Hierarchy. *Tech Report CSD-930021*, 7 pages, UCLA, 1993.

[30] Borowsky E., Gafni E., Lynch N. and Rajsbaum S., The BG Distributed Simulation Algorithm. *Distributed Computing*, 14(3) :127-146, 2001.

[31] Burns, J., Symmetry in Systems of Asynchronous Processes. *22nd IEEE Symposium on Foundations of Computer Science (FOCS'81)*, IEEE Computer Press, 169-174, 1981.

[32] Castañeda A. and Rajsbaum S., New Combinatorial Topology Upper and Lower Bounds for Renaming. *Proc. 27th ACM Symposium on Principles of Distributed Computing (PODC'08)*, ACM Press, pp. 295-304, Toronto (Canada), 2008.

[33] Castañeda A., A Study of the Wait-free Solvability of Weak Symmetry Breaking and Renaming. PhD Thesis, Posgrado en Ciencia e Ingeniería de la Computación, UNAM, Mexico, December 2010.

[34] Castañeda A., Imbs D., Rajsbaum S. and Raynal M., Renaming is Weaker than Set Agreement but for Perfect Renaming : A Map of Sub-Consensus Tasks. *Proceedings of the 10th International Latin American Symposium on Theoretical Informatics (LATIN 2012)*, Springer-Verlag LNCS, 2012.

[35] Castañeda A., Rajsbaum S. and Raynal M., The Renaming Problem in Shared Memory Systems : an Introduction. *Tech Report 1960*, IRISA, Université de Rennes (F), 29 pages, 2010. Submitted to publication.

[36] Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2) :225-267, 1996.

[37] Chaudhuri S., More *Choices* Allow More *Faults :* Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation,* 105 :132-158, 1993.

[38] Chandra T., Hadzilacos V. and Toueg S. : The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4) :685-722, 1996.

[39] Chaudhuri S. and Reiners P., Understanding the Set Consensus Partial Order Using the Borowsky-Gafni Simulation. *Proc. 10th Int'l Workshop on Distributed Algorithms (WDAG'96, now DISC)*, Springer Verlag LNCS #1151, pp. 362-379, 1996.

[40] Chandy K.M. and Lamport L., Distributed Snapshots : Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1) :63-75, 1985.

[41] Crain T., Imbs D. and Raynal M., Read invisibility, virtual world consistency and probabilistic permissiveness are compatible. *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2011)*, Springer-Verlag LNCS #7016, pp. 244-257, 2011.

[42] Crain T., Imbs D. and Raynal M., Towards a universal construction for transaction-based multiprocess programs. *Proceedings of the 13th International Conference on Distributed Computing and Networking (ICDCN 2012)*, Springer-Verlag LNCS #7129, pp. 61-75, 2012.

[43] Delporte-Gallet C., Fauconnier H. and Guerraoui R., Tight Failure Detection Bounds on Atomic Object Implementations. *Journal of the ACM*, 57(4), Article 22, 32 pages, 2010.

[44] Delporte-Gallet C., Fauconnier H., Guerraoui R., Hadzilacos V., Kouznetsov P. and Toueg S., The Weakest Failure Detectors to Solve Certain Fundamental Problems in Distributed Computing. *Proc. 23th ACM Symposium on Principles of Distributed Computing (PODC'04)*, ACM Press, pp. 338-346, 2004.

[45] Delporte-Gallet C., Fauconnier H., Guerraoui R. and Tielmann A., The disagreement power of an adversary. *Distributed Computing,* 24(3-4) :137-–147, 2011.

[46] Dijkstra, E.W., Solution of a Problem in Concurrent Programming Control. *Communications of the ACM,* 8(9) :569, 1965.

[47] Dolev D., Lynch N., Pinter S., Stark E. and Weihl W. Reaching Approximate Agreement in the Presence of Faults. *Journal of the ACM*, 33(3) :499–516, 1986.

[48] Dinitz Y., Moran S. and Rajsbaum S., Bit complexity of Breaking and Achieving Symmetry in Chains and Rings. *Journal of the ACM*, 55(1), article 3, 32 pages, 2008.

[49] Ellen F., How Hard Is It to Take a Snapshot ? *Proc. 31th Conference on Current Trends in Theory and Practice of Computer Science (SOFTSEM'05)*, Springer-Verlag #3381, pp. 28-37, 2005.

[50] Felber P., Fetzer C., Guerraoui R. and Harris T., Transactions Are Back, But Are They the Same ? *ACM SIGACT New, Distributed Computing Columns*, 39(1) : 47-58, 2008.

[51] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *JACM*, 32(2) :374-382, 1985.

[52] Gafni E., Round-by-round Fault Detectors : Unifying Synchrony and Asynchrony. *Proc. 17th ACM Symposium on Principles of Distributed Computing (PODC'98)*, ACM Press, pp. 143-152, 1998.

[53] Gafni E., Renaming with k-set Consensus : an Optimal Algorithm in $n + k - 1$ Slots. *Proc. 10th Int'l Conference On Principles Of Distributed Systems (OPODIS'06)*, Springer Verlag LNCS #4305, pp. 36-44, 2006.

[54] Gafni E., The 01-Exclusion Families of Tasks. *Proc. 12th In'l Conference on Principles of Distributed Systems (OPODIS'08*, Springer Verlag LNCS #5401, pp. 246-258, 2008.

[55] Gafni E., The Extended BG Simulation and the Characterization of t-Resiliency. *Proc. 41th ACM Symposium on Theory of Computing (STOC'09)*, ACM Press, 2009.

[56] Gafni E. and Guerraoui R., Simulating Few by Many. *Unpublished manuscript*, 2009, http ://www.cs.ucla.edu/ eli/eli/kconc.pdf

[57] Gafni E. and Kuznetsov P., N-Consensus is the Second Strongest Object for N+1 Processes. *Proc. 11th Int'l Conference On Principle Of Distributed Systems (OPODIS 2007)*, Springer Verlag LNCS #4878, pp. 260-273, 2007.

[58] Gafni E. and Kuznetsov P., On Set Consensus Numbers. *Proc. 23th Int'l Symposium on on Distributed Computing (DISC'09)*, Springer Verlag LNCS #5805, pp. 35-47, 2009.

[59] Gafni E., Mostéfaoui A., Raynal M., Travers C., From Adaptive Renaming to Set Agreement. *Theoretical Computer Science*, 410 :1328-1335, 2009.

[60] Gafni E. and Rajsbaum S., Musical Benches. *19th International Symposium on Distributed Computing (DISC'05)*, Springer Verlag LNCS #3724, pp. 63-77, 2005.

[61] Gafni E., Rajsbaum S. and Herlihy M., Subconsensus Tasks : Renaming is Weaker than Set Agreement. *Proc. 20th Int'l Symposium on Distributed Computing (DISC'06)*, LNCS #4167, pp.329-338, 2006.

[62] Gafni E., Rajsbaum S., Raynal M. and Travers C., The Committee Decision Problem. *Proc. Latin American Theoretical Informatics Symposium (LATIN'06)*. Springer Verlag LNCS #3887, pp. 502-514, 2006.

[63] Gafni E., Raynal M. and Travers C., Test&set, Adaptive Renaming and Set Agreement : a Guided Visit to Asynchronous Computability. *26th IEEE Symposium on Reliable Distributed Systems (SRDS'07)*, IEEE Computer Press, pp. 93-102, 2007.

[64] Gifford D.K., Weighted Voting for Replicated Data. *Proc. 7th ACM Symposium on Operating System Principles (SOSP'79)*, ACM Press, pp. 150-172, 1979.

[65] Guerraoui R. and Kapałka M., On the Correctness of Transactional Memory. *Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, ACM Press, pp. 175-184, 2008.

[66] Guerraoui R., Kapałka M. and Kouznetsov P., The Weakest Failure Detectors to Boost Obstruction-Freedom. *Distributed Computing*, 20(6) :415-433, 2008.

[67] Guerraoui R. and Kuznetsov P., Failure Detectors as Type Boosters. *Distributed Computing*, 20 :343-358, 2008.

[68] Guerraoui R. and Raynal M., From Unreliable Objects to Reliable Objects : the Case of Atomic Registers and Consensus. *9th Int'l Conference on Parallel Computing Technologies (PaCT'07)*, Springer LNCS #4671, pp. 47-61, 2007.

[69] Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1) :124-149, 1991.

[70] Herlihy M.P. and Luchangco V., Distributed Computing and the Multicore Revolution. *ACM SIGACT News, Distributed Computing Column*, 39(1) : 62-72, 2008.

[71] Herlihy M.P., Luchangco V. and Moir M., Obstruction-free synchonization : double-ended queues as an example. *Proc. 23th Int'l IEEE Conference on Distributed Computing Systems*, pp. 522-529, 2003.

[72] Herlihy M. and Rajsbaum S. The Decidability of Distributed Decision Tasks. *Proc. 29th ACM Symposium on Theory of Computing (STOC'97)*, ACM Press, pp. 589-598, 1997.

[73] Herlihy M.P. and Rajsbaum S., Algebraic Spans, *Mathematical Structures in Computer Science*, 10(4) : 549-573, 2000.

[74] Herlihy M. and Rajsbaum S. A Classification of Wait-Free Loop Agreement Tasks. *Theoretical Computer Science*, 291(1) :55-77, 2003.

[75] Herlihy M., Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6) :858-923, 1999.

[76] Herlihy M.P. and Shavit N., The Art of Mutiprocessor Programming. *Morgan Kaufmann Pub.*, San Francisco (CA), 508 pages, 2008.

[77] Herlihy M.P. and Wing J.L., Linearizability : a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3) :463-492, 1990.

[78] Imbs D. and Raynal M., Help when needed, but no more : Efficient Read/Write Partial Snapshot. *Journal of Parallel and Distributed Computing* 72(1), Elsevier, pp. 1-12, 2012.

[79] Imbs D. and Raynal M., Virtual world consistency : a condition for STM systems (with a versatile protocol with invisible read operations). *Theoretical Computer Science*, Elsevier, To appear, 2011.

[80] Imbs D. and Raynal M., A liveness condition for concurrent objects : x-wait-freedom. *Concurrency and Computation : Practice and Experience* 23(17), Wiley, pp. 2154-2166, 2011.

[81] Imbs D. and Raynal M., Software transactional memories : an approach for multicore programming. *Journal of Supercomputing* 57(2), Springer, pp. 203-215, 2011.

[82] Imbs D., Rajsbaum S. and Raynal M., The Universe of Symmetry Breaking Tasks. *Proceedings of the 18th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2011)*, Springer-Verlag LNCS #6796, pp. 66-77, 2011.

[83] Imbs D. and Raynal M., The weakest failure detector to implement a register in asynchronous systems with hybrid communication. *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2011)*, Springer-Verlag LNCS #6976, pp. 268-282, 2011.

[84] Imbs D. and Raynal M., A Simple Snapshot Algorithm for Multicore Systems. *Proceedings of the 5th Latin-American Symposium on Dependable Computing (LADC 2011)*, IEEE Computer Press, pp. 17-24, 2011.

[85] Imbs D. and Raynal M., The Multiplicative Power of Consensus Numbers. *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing (PODC 2010)*, ACM Press, pp. 26-35, 2010.

[86] Imbs D. and Raynal M., The Multiplicative Power of Consensus Numbers. *Tech Report #1943*, IRISA, Univ. de Rennes 1, 2010.

[87] Imbs D., Raynal M. and Taubenfeld G., On Asymmetric Progress Conditions. *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing (PODC 2010)*, ACM Press, pp. 55-64, 2010.

[88] Imbs D. and Raynal M., The x-Wait-freedom Progress Condition. *Proceedings of the 16th International Euro-Par Conference (Euro-Par 2010)*, Springer Verlag LNCS #6271, pp. 584-595, 2010.

[89] Imbs D. and Raynal M., On Adaptive Renaming under Eventually Limited Contention. *Proceedings of the 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2010)*, Springer Verlag LNCS #6366, pp. 377-387, 2010.

[90] Imbs D. and Raynal M., Visiting Gafni's Reduction Land : From the BG Simulation to the Extended BG Simulation. *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2009)*, Springer Verlag LNCS #5873, pp. 369-383, 2009.

[91] Imbs D. and Raynal M., Help When Needed, But No More : Efficient Read/Write Partial Snapshot. *Proceedings of the 23rd International Symposium on Distributed Computing (DISC 2009)*, Springer-Verlag, LNCS #5805, pp. 142-156, 2009.

[92] Imbs D. and Raynal M., Software Transactional Memories : An Approach for Multicore Programming. *Proceedings of the 10th International Conference on Parallel Computing Technologies (PaCT 2009)*, Springer-Verlag, LNCS #5698, pp. 26-40, 2009.

[93] Imbs D. and Raynal M., A Versatile STM Protocol with Invisible Read Operations That Satisfies the Virtual World Consistency Condition. *Proceedings of the 16th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2009)*, Springer-Verlag, LNCS #5869, pp. 266-280, 2009.

[94] Imbs D. and Raynal M., Trying to Unify the LL/SC Synchronization Primitive and the Notion of a Timed Register. *Proceedings of the 26th IEEE International Conference on Advanced Information Networking and Applications (AINA-2012)*, IEEE Computer Press, 2012.

[95] Inoue I., Chen W., Masuzawa T. and Tokura N., Linear Time Snapshots Using Multi-writer Multi-reader Registers. *Proc. 8th Int'l Workshop on Distributed Algorithms (WDAG'94)*, Springer-Verlag #857, pp. 130-140, 1994.

[96] Israeli A. and Rappoport L., Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. *Proc. 13th ACM Symposium on Principles of Distributed Computing (PODC'94)*, pp. 151-160, 1994.

[97] Jayanti P., An Optimal Multiwriter Snapshot Algorithm. *Proc. 37th ACM Symposium on Theory of Computing (STOCS'05)*, ACM Press, pp. 723-732, 2005.

[98] Jayanti, P., and Toueg, S. Wakeup under Read/Write Atomicity. *Proc. 4th Int'l Workshop on Distributed Algorithms (WDAG'90)*, Springer Verlag LNCS #486, pp. 277–288, 1990.

[99] Lamport. L., On Interprocess Communication, Part 1 : Basic formalism, Part II : Algorithms. *Distributed Computing*, 1(2) :77-101,1986.

[100] Lamport. L., The Part-time Parliament. *ACM Transactions on Computer Systems*, 16(2) :133-169, 1998.

[101] Loui M.C., and Abu-Amara H.H., Memory Requirements for Agreement Among Unreliable Asynchronous Processes. *Par. and Distributed Computing : vol. 4 of Advances in Comp. Research,* JAI Press, 4 :163-183, 1987.

[102] Lynch N.A., Distributed Algorithms. *Morgan Kaufmann Pub.*, San Francisco (CA), 872 pages, 1996.

[103] Moran, S., and Wolfsthal, Y., An extended Impossibility Result for Asynchronous Complete Networks. *Information Processing Letters* 26 :141-151, 1987.

[104] Mostefaoui M., Raynal M., and Travers C., Exploring Gafni's Reduction Land : from Ω^k to Wait-free adaptive $(2p - \lceil \frac{p}{k} \rceil)$-renaming via k-set Agreement. *Proc. 20th Int'l Symposium on Distributed Computing (DISC'09)*, Springer Verlag LNCS #4167, pp. 1-15, 2006.

[105] Mostéfaoui A., Raynal M. and Travers C., Narrowing Power vs Efficiency in Synchronous Set Agreement : Relationship, Algorithms and Lower Bound. *Theoretical Computer Science*, 411 :58-69, 2010.

[106] Neiger, G., Failure Detectors and the Wait-free Hierarchy. *Proc. 14th ACM Symposium on Principles of Distributed Computing (PODC'95)*, ACM Press, pp. 100-109, 1995.

[107] Papadimitriou Ch.H., The Serializability of Concurrent Updates. *Journal of the ACM*, 26(4) :631-653, 1979.

[108] Powell D., Failure Mode Assumptions and Assumption Coverage. *22nd Int'l Symposium on Fault-Tolerant Computing (FTCS-22)*, IEEE Computer Society Press, pp.386-395, 1992.

[109] Raynal M., Locks Considered Harmful : a Look at Non-traditional Synchronization. *Proc. 6th Int'l Workshop on Software Technologies for Future Embedded and Ubiquitous Computing Systems (SEUS'08)*, Springer-Verlag, #LNCS 5287, pp. 369-380, 2008.

[110] Raynal M., Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems. *Morgan & Claypool Pub.*, 251 pages, 2010 (ISBN 978-1-60845-293-4).

[111] Raynal M. and Travers C., In Search of the Holy Grail : Looking for the Weakest Failure Detector for Wait-free Set Agreement. *Proc. 10th Int'l Conference on Principles of Distributed Systems (OPODIS'06)*, Springer Verlag LNCS #4305, pp. 1-17, 2006.

[112] Saks M. and Zaharoglou F., Wait-Free k-Set Agreement is Impossible : The Topology of Public Knowledge. *SIAM Journal on Computing*, 29(5) :1449-1483, 2000.

[113] Styer E., and Peterson G. L., Tight Bounds for Shared Memory Symmetric Mutual Exclusion Problems. *Proc. 8th ACM Symposium on Principles of Distributed Computing (PODC'89)*, ACM Press, pp. 177-192, 1989.

[114] Shavit N. and Touitou D., Software Transactional Memory. *Distributed Computing*, 10(2) :99-116, 1997.

[115] Taubenfeld G., Synchronization Algorithms and Concurrent Programming. *Pearson Prentice-Hall*, ISBN 0-131-97259-6, 423 pages, 2006.

[116] Taubenfeld G., Contention-Sensitive Data Structure and Algorithms. *Proc. 23th Int'l Symposium on Distributed Computing (DISC'09)*, Springer Verlag LNCS #5805, pp. 157-171, 2009.

[117] Taubenfeld G., On the Computational Power of Shared Objects. *Proc. 13th Int'l Conference On Principle Of Distributed Systems (OPODIS 2009)*, Springer Verlag LNCS #5923, pp. 270-284, 2009.

[118] Taubenfeld G., The Computational Structure of Progress Conditions. *Proc. 24th Int'l Symposium on Distributed Computing (DISC'10)*, Springer Verlag LNCS #6343, pp. 221-235, 2010.

VU :

Le Directeur de Thèse
(Nom et Prénom)

VU :

Le Responsable de l'École Doctorale

VU pour autorisation de soutenance

Rennes, le

Le Président de l'Université de Rennes 1

Guy CATHELINEAU

VU après soutenance pour autorisation de publication :

Le Président de Jury,
(Nom et Prénom)