Auto-stabilisation Efficace

Sébastien Tixeuil

Orsay
N° d'ordre: 6037

THÈSE

présentée à l'Université Paris XI Orsay
pour obtenir le grade de Docteur en Sciences
spécialité Informatique

—

Auto-stabilisation Efficace

Sébastien TIXEUIL

—

soutenue le 14 janvier 2000 devant le jury composé de

M.	Joffroy BEAUQUIER	Directeur
M.	Dominique GOUYOU-BEAUCHAMPS	Président
M.	Christian LAVAULT	Examinateur
M.	Michel RAYNAL	Rapporteur
M.	Vincent VILLAIN	Rapporteur

Table des matières

1 Présentation du Domaine **7**
- 1.1 Systèmes Répartis . . . 7
  - 1.1.1 Motivations . . . 8
  - 1.1.2 Spécificités . . . 9
  - 1.1.3 Hypothèses Pertinentes . . . 10
  - 1.1.4 Problèmes Classiques . . . 13
- 1.2 Tolérance aux défaillances . . . 15
  - 1.2.1 La tolérance aux pannes . . . 15
  - 1.2.2 L'auto-stabilisation. . . . 16
  - 1.2.3 Approches Multi-tolérantes . . . 17
- 1.3 Critères d'efficacité . . . 18
- 1.4 Résumé . . . 19

2 Modèle **21**
- 2.1 Références . . . 21
- 2.2 Préliminaires . . . 22
  - 2.2.1 Graphes . . . 22
  - 2.2.2 Automates et Produit Synchronisé . . . 24
- 2.3 Eléments de Base . . . 28
  - 2.3.1 Liens . . . 28
  - 2.3.2 Processeurs . . . 30
- 2.4 Systèmes Répartis . . . 32
  - 2.4.1 Définitions . . . 32
  - 2.4.2 Projections des Exécutions . . . 34
  - 2.4.3 Equité . . . 36
  - 2.4.4 Démons . . . 37
  - 2.4.5 Spécification . . . 39
- 2.5 Auto-stabilisation . . . 41
  - 2.5.1 Définition . . . 41
  - 2.5.2 Prouver l'Auto-stabilisation . . . 42
- 2.6 Complexité . . . 43
  - 2.6.1 Complexité en Espace . . . 43

2.6.2 Complexité en Temps . . . 44
2.7 Résumé . . . 46

3 Détecter les Défaillances **49**
3.1 Références . . . 49
3.2 Hypothèses Spécifiques au Chapitre . . . 50
3.3 Détecteurs de Défaillances Transitoires . . . 50
3.4 Détecteurs de Défaillances pour les Tâches Statiques . . . 54
3.4.1 Tâches Globales . . . 54
3.4.2 Tâches Locales . . . 60
3.4.3 Autres Tâches Statiques . . . 62
3.5 Détecteurs de Défaillances pour les Tâches Dynamiques . . . 66
3.5.1 Tâches Bornées . . . 67
3.5.2 Tâches Déterministes et Non-interactives . . . 70
3.5.3 Tâches Equitables . . . 70
3.6 Détecteurs de Défaillances pour les Algorithmes . . . 71
3.7 Implantation . . . 72
3.8 Résumé . . . 73

4 Auto-stabilisation sans Surcoût **75**
4.1 Références . . . 75
4.2 Hypothèses Spécifiques au Chapitre . . . 77
4.2.1 Graphes Orientés Acycliques . . . 77
4.2.2 r-Opérateurs . . . 78
4.3 Algorithme Paramétré . . . 81
4.3.1 Description Informelle . . . 81
4.3.2 Algorithme . . . 82
4.4 Preuve de Correction . . . 83
4.4.1 Atomicité de Lecture/Ecriture . . . 83
4.4.2 Cas des Graphes sans Cycles . . . 84
4.4.3 Cas Général . . . 87
4.4.4 Complexité . . . 97
4.5 Applications . . . 99
4.5.1 Tri Topologique sur des Graphes sans Cycles . . . 99
4.5.2 Calcul de Distance, Arbre et Forêt de Longueur Minimale . . . 101
4.5.3 Arbre et Forêt de plus Courts Chemins . . . 103
4.5.4 Arbre et Forêt des plus Fiables Émetteurs . . . 105
4.5.5 Arbre en Profondeur . . . 107
4.6 Cas des Ensembles Infinis . . . 109
4.7 Résumé . . . 111

5 Synchronisation **113**
5.1 Références . . . 113
5.2 Hypothèses Spécifiques au Chapitre . . . 115
5.3 Le Synchroniseur Global . . . 115
5.3.1 Spécification du Problème . . . 115
5.3.2 Description Informelle . . . 116
5.3.3 Algorithme . . . 118
5.3.4 Preuve de Correction . . . 120
5.3.5 Complexité . . . 122
5.4 Le Synchroniseur de Voisinage . . . 124
5.4.1 Spécification du Problème . . . 124
5.4.2 Description Informelle . . . 124
5.4.3 Algorithme . . . 124
5.4.4 Preuve de Correction . . . 125
5.4.5 Complexité . . . 127
5.5 Applications . . . 127
5.5.1 Horloge Globale . . . 128
5.5.2 Fonction Globale . . . 129
5.5.3 Tri Réparti . . . 129
5.5.4 Diffusion Parallèle . . . 130
5.6 Résumé . . . 137

6 Communications à Délai Borné **139**
6.1 Références . . . 139
6.2 Hypothèses Spécifiques au Chapitre . . . 141
6.2.1 Temps de Communication . . . 141
6.2.2 Modèle de Communication . . . 142
6.2.3 Opérations de Base . . . 147
6.3 Construction d'un Circuit Eulérien . . . 150
6.3.1 Suppression des Messages . . . 150
6.3.2 Circuit Eulérien . . . 153
6.4 Applications . . . 159
6.4.1 Election d'un Chef . . . 159
6.4.2 Recensement . . . 165
6.5 Résumé . . . 172

Introduction

> I regard this [Self-stabilization] as Dijkstra's most brilliant work — at least, his most brilliant published paper. [...] I regard it to be a milestone in work on fault-tolerance.
>
> Leslie Lamport

Si les systèmes répartis auto-stabilisants — dont il est question dans cette thèse — ont fait l'objet d'études qui s'inscrivent dans le cadre technique de l'Informatique, il en existe toutefois de nombreux exemples dans notre environnement quotidien.

Imaginons un instant que pour fêter le passage à l'an 2000, l'université de Paris Sud fasse appel à une société de services pour monter un spectacle son et lumières. Le jour dit, une grande toile blanche est tendue sur la façade du Laboratoire de Recherche en Informatique, et une équipe composée de projectionnistes et d'un ingénieur du son s'installe sur le parking. La société de services a laissé à l'ingénieur du son une platine à disques compacts, un disque contenant 5 plages sonores ainsi que les instructions suivantes : jouer la plage 2 du disque compact, puis la plage 5, la plage 1, puis les plages 3 et 4 deux fois de suite, puis reprendre à la plage 2 et recommencer la même procédure. De la même manière, chaque projectionniste dispose d'un projecteur de diapositives, d'un bac de diapositives déjà positionnées, et des instructions suivantes : passer chaque diapositive l'une après l'autre, en laissant une minute entre chaque changement d'image, puis recommencer à partir de la première diapositive. Étant donnée la grande taille de la toile, les projecteurs de diapositives sont alignés et réglés de manière à former une image unique — un mur d'images — à partir des images projetées par chaque appareil. De façon à ne pas géner les spectateurs, tous les projectionnistes se trouvent placés dans une fosse, d'où ils ne peuvent voir la mosaïque projetée. Enfin, de manière à se prémunir contre d'éventuelles coupures de courant, un groupe électrogène est disposé à proximité.

Avec le vocabulaire des systèmes répartis et de l'auto-stabilisation, chaque projectionniste ou ingénieur du son devient un *processeur*. Le comportement de chaque personne est régi par les instructions que la société lui a laissées (son *programme*) et par sa position courante dans ces instructions (son *état*). L'ensemble que nous avons décrit est un *système réparti*, c'est à dire une collection de processeurs indépendants (leur programme et leur état leur sont propres) qui collaborent à une même tâche : un spectacle son et lumières qui apparaîsse satisfaisant pour les spectateurs.

Évidement, aux douze coups de minuit, une coupure de courant survient, mettant en fonction le groupe électrogène. Malheureusement, à la fois la platine de disques compacts et les projecteurs supportent mal les brusques variations de courant qui viennent de survenir : la platine joue maintenant la plage 1 du disque compact ; et chaque projecteur a avancé de plusieurs diapositives, de manière que le mur d'images n'affiche plus qu'une image incohérente. L'ingénieur du son décide de continuer à exécuter les instructions de son manuel comme si rien ne s'était passé : il enchaîne avec les plages 3 et 4. De leur coté, les projectionnistes aimeraient retrouver une image cohérente à afficher, mais ne savent pas de quelle manière s'y prendre. En effet, étant donné l'alignement des projecteurs et le manque d'éclairage, chaque projectionniste ne peut consulter le numéro de diapositive projetée que sur son propre projecteur ou sur celui de ses voisins immédiats.

Cet arrêt puis reprise du service avec perturbations peut être assimilé à une *défaillance transitoire* de notre système réparti qui présente plusieurs particularités : le programme de chaque processeur (c'est à dire les instructions laissées aux membres du personnel) n'a pas changé, à la différence de son état (c'est à dire la position courante dans la suite d'instructions) qui lui a été perturbé.

Comme l'interruption de courant n'est que momentanée, il serait dommage de ne pas être en mesure de rétablir un service correct, c'est à dire la resynchronisation de la mosaïque d'images, en un temps acceptable pour les spectateurs. Si chaque projectionniste se borne à continuer à suivre les instructions, le mur d'images peut rester indéfiniment incohérent.

Présentons maintenant une solution à ce problème, en donnant des instructions supplémentaires à chaque projectionniste : avant de changer une diapositive, vérifier que le numéro de la diapositive que je me prépare à afficher est inférieur ou égal au numéro de la diapositive affichée par chacun de mes voisins, et si cette vérification s'avère fausse, alors afficher la diapositive dont le numéro est égal au plus petit numéro affiché par mes voisins.

Donnons maintenant quelques arguments pour nous assurer que ces nouvelles directives suffisent à rétablir un fonctionnement correct. Si le nombre de diapositives est suffisamment grand pour qu'aucun projectionniste ne retourne afficher la première diapositive, alors cette procédure permet de resynchroniser le mur d'images. En effet, il est possible de déterminer un projectionniste dont la diapositive actuellement affichée est d'indice minimal. Alors ses deux voisins vont à leur tour afficher ce numéro de diapositive. Une fois les voisins synchronisés, les voisins des voisins les rejoignent à leur tour, jusqu'à ce que tout le monde soit synchronisé. Si certains projectionnistes, malgré le grand nombre de diapositives à afficher, sont revenus à la projection de la première diapositive, alors un tel raisonnement ne tient plus. Dans ce cas, on peut faire le raisonnement suivant : chaque projectionniste soit est proche d'un projectionniste affichant une image d'indice peu élevé, auquel cas il diminue l'indice de sa propre image, soit il était déjà en train d'afficher l'une des dernière images, auquel cas il recommence à afficher les premières images. Dans tous les cas, et en supposant que le nombre de diapositives à projeter est suffisamment grand, après un temps relativement court, tous les projectionnistes affichent une image d'indice identique et qui se trouve parmi les premières images.

Le résultat des directives additionnelles exécutées par chacun des projectionnistes est que la mosaïque d'images est de nouveau cohérente, pour le plus grand plaisir des spectateurs. Autrement dit, la procédure que nous avons ajoutée rend le système réparti *auto-stabilisant* pour la spécification de la mosaïque d'images : *indépendamment de l'état global initial* (les indices des images affichées par les projectionnistes sont quelconque suite à la coupure momentanée de courant), *le système réparti converge de lui-même* (il n'est nul besoin de l'intervention d'un cadre de la société afin de remettre bon ordre à la mosaïque) *vers un comportement correct* (la mosaïque d'images est de nouveau cohérente et le reste en l'absence de nouvelles défaillances).

Il va de soi que les instructions supplémentaires (vérifier que l'indice de l'image courante est minimal parmi mes voisins) exécutées localement ne sont pas sans incidence sur le spectacle : chaque projectionniste doit non seulement vérifier que, toutes les minutes, l'image qu'il a la charge de projeter change, mais il doit aussi vérifier que ses voisins agissent de concert avec lui. Au bout du compte, il se peut que, suite à ces vérification systématiques, le mur d'images ne soit plus rafraîchi toutes les minutes mais toutes les deux minutes.

L'échange d'informations effectué systématiquement afin de pallier les possibles défaillances transitoires est apellé le *surcoût* engendré par rapport à un système qui propose le même service mais qui ne tolère pas de défaillances (un système réparti non-stabilisant). Dans cette thèse, nous nous attachons principalement à minimiser ce surcoût de manière à ce que des solutions auto-stabilisantes puissent être utilisées en lieu et place de solutions non tolérantes aux défaillances transitoires sans que les performances s'en trouvent dégradées de manière inacceptable.

Le chapitre 1 (Présentation du Domaine) introduit informellement les systèmes répartis informatiques, l'auto-stabilisation et les critères d'efficacité que nous considérons. Ces concepts sont formalisés au chapitre 2 (Modèle), qui fournit un cadre général aux résultats présentés dans la suite.

Le chapitre 3 (Détecter les défaillances) étend les résultats présentés dans [BDDT98] et présente une classification de différents problèmes suivant la localité nécessaire à la détection d'une erreur. Dans l'exemple précédent, détecter une erreur se fait avec une localité en distance de un : il suffit de regarder l'indice de l'image affichée par ses voisins pour déterminer que tous les projectionnistes n'affichent pas la même image. Par contre, s'assurer qu'à tout instant, un unique projectionniste affiche l'image numéro 1, nécessite qu'au moins un des projectionnistes ait une connaissance globale des indices des images affichées. Dans ce dernier cas, la localité en distance est proportionelle à la moitié des projectionnistes présents : les projectionnistes formant une chaîne, le projectionniste situé au milieu a alors une vue globale sur tous les autres.

Les chapitres 4 (Auto-stabilisation sans Surcoût) et 5 (Synchronisation) présentent deux voies possibles pour minimiser le surcoût dans le cas des tâches statiques (le système réparti doit aboutir à un état global qui finit par ne plus varier, comme une mosaïque d'images qui

resterait figée une fois que l'image est cohérente) et dynamiques (le système aboutit à un comportement cohérent, comme dans l'exemple développé plus haut), respectivement.

Le chapitre 4 complète certains résultats publiés dans [DT98b, DT99, DDT99] et propose une *condition* sur les instructions données à chacun des processeurs du système garantissant que le système est auto-stabilisant. Le fait d'utiliser une condition sur le programme local indique que l'auto-stabilisation du système se fait *sans surcoût* (le programme local n'est pas modifié pour détecter et corriger d'éventuelles défaillances transitoires), et reste indépendante de l'agencement du système (les processeurs peuvent agir à des vitesses différentes, les possibilités de communication sont arbitraires, aucune connaissance globale n'est nécessaire).

Le chapitre 5 unifie les résultats présentés dans [ABC$^+$98, ABDT98, JADT99] et rend compte de l'idée intuitive que si un processeur est distingué dans le système (et considéré par les autres comme un chef-d'orchestre), alors il est beaucoup plus facile de construire des systèmes auto-stabilisants. Dans l'exemple des projectionnistes, si l'ingénieur du son en charge de la musique est suivi par tous les projectionnistes comme un élément de référence, alors il suffit d'établir une correspondance entre les mesures des morceaux de musique et les indices des diapositives à projeter pour qu'en cas de défaillances transitoires, tout le système converge rapidement vers un comportement correct. Chaque projectionniste ne passe plus son temps à regarder ses voisins, mais se borne à écouter la musique diffusée. A chaque nouvelle mesure reconnue, il affiche la diapositive adéquate. Comme la musique diffusée est la même partout, tous les projectionnistes affichent leurs images à l'unisson. Un tel mécanisme est appelé un *synchroniseur*, et permet l'unification de l'exécution d'un programme local sur plusieurs processeurs. Le chapitre 5 propose deux synchroniseurs aux propriétés différentes, ainsi que plusieurs applications.

Enfin, le chapitre 6 (Communications à Délai Borné) reprend une partie des résultats publiés dans [TB96, BDT99, BKT99] et propose l'utilisation d'une technique de communication plus efficace dans les systèmes auto-stabilisants, de manière à minimiser le surcoût dû à la transmission d'informations. Habituellement, les communications dans les systèmes répartis auto-stabilisants se font de manière indivisible tout comme dans le système postal : une information est empaquetée dans une enveloppe et transmise en un seul bloc. Quand les deux processeurs qui s'échangent des informations ne sont pas des voisins immédiats, chacun des processeurs servant d'intermédiaire doit réceptionner l'enveloppe, constater que le message ne lui est pas destiné, avant de réexpédier le message enveloppé. Le système de communication développé au chapitre 6 est plus proche du système du télégraphe : une information est une suite d'informations élémentaires qui suivent le même chemin. Ce faisant, quand une information doit être acheminée à un destinataire en passant par plusieurs processeurs intermédiaires, le délai sur chacun d'eux est réduit puisqu'il peut commencer à retransmettre l'information avant de l'avoir complètement reçue.

A titre de résumé, la figure 1 propose un plan de lecture de la thèse.

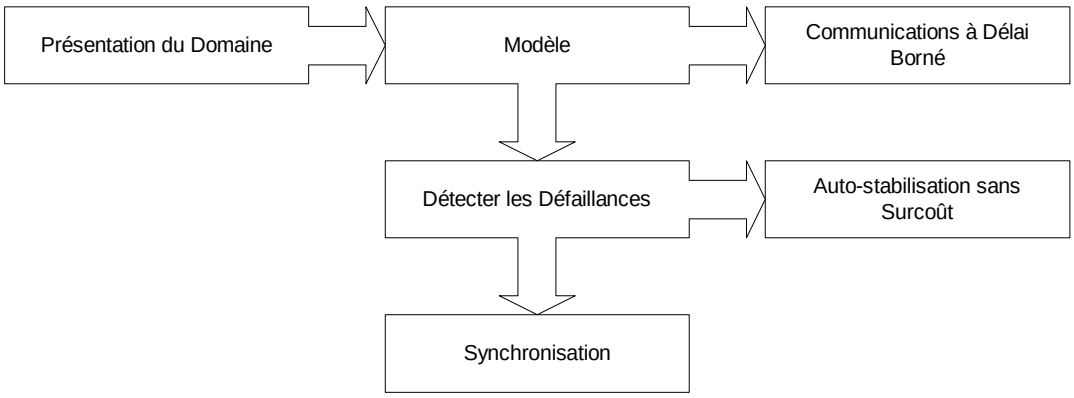

FIG. 1 – Plan de lecture.

Chapitre 1

Présentation du Domaine

L'étude menée dans le cadre de cette thèse relève du domaine des algorithmes auto-stabilisants, qui sont des algorithmes répartis tolérant certains types de pannes transitoires. Le problème précis auquel nous nous sommes intéressés est la maximalisation de l'efficacité de ces algorithmes suivant plusieurs critères.

Dans ce chapitre, après avoir rappelé quelques caractéristiques générales des systèmes répartis, nous décrirons informellement les algorithmes répartis et l'auto-stabilisation. La plupart des aspects présentés dans ce chapitre peut être retrouvés de manière plus détaillée dans [Tel94] et dans [Lav95].

1.1 Systèmes Répartis

Les systèmes répartis sont les systèmes qui gèrent des *processeurs* (aussi appelés *sites*, *machines*, *ordinateurs* ou *nœuds*) reliés entre eux par des *liens de communication* (aussi appelés *canaux*, *câbles*, *registres*, *arcs* ou *arêtes*). Ces processeurs ont la possiblité de s'échanger des *informations* (aussi appelées *messages*) par l'intermédiaire des liens de communication.

Les systèmes répartis existent sous plusieurs formes *a priori* dissemblables mais qui possèdent plusieurs caractéristiques en commun. Par exemple, un ordinateur exécutant un système d'exploitation *multi-processus* (comme Unix) peut être considéré comme un système réparti : le système supporte *simultanément* plusieurs programmes en cours d'exécution, et ces programmes sont capables de communiquer les uns avec les autres. De la même manière, une machine parallèle constituée de plusieurs processeurs semblables et partageant une mémoire commune est également un système réparti : chaque processeur exécute un programme et communique avec les autres processeurs *via* la mémoire partagée. Un troisième exemple, celui des *réseaux d'ordinateurs*, regroupe un grand nombre d'ordinateurs individuels (de caractéristiques différentes et exécutant des systèmes d'exploitation différents) liés entre eux par divers supports (câbles, ondes électromagnétiques), et qui communiquent au moyen d'un langage commun : le *protocole réseau*.

1.1.1 Motivations

Historiquement, le développement des systèmes répartis tels que nous les connaissons aujourd'hui est lié à plusieurs besoins :

1. **Communiquer** : Le besoin de communiquer des textes, des programmes, des images, du son, et même aujourd'hui des données vidéo a conduit au développement des réseaux d'ordinateurs à grande distance. Les universités et principales entreprises ont initié le mouvement dans les années soixante, alors qu'elles venaient de s'équiper d'un ordinateur central. Les échanges de données multimédia sont aujourd'hui monnaie courante entre des ordinateurs individuels interconnectés.

2. **Economiser :** Alors que le coût des micro-ordinateurs individuels ne cesse de baisser, beaucoup de périphériques comme les imprimantes et les unités de sauvegarde sont restés relativement coûteux. Le développement des réseaux locaux d'ordinateurs est dû en grande partie à la mise en place d'un système de partage des éléments les plus coûteux par les éléments les moins coûteux. En plus de générer des coûts moindres par rapport à un système centralisé, cette façon de procéder permet une bonne extensibilité en cas d'avancées technologiques par l'ajout progressif de composants.

3. **Accélérer :** Le constat général qu'une tâche est plus rapidement réalisée quand plus de ressources lui sont affectées a permis de développer les systèmes répartis constitués de machines parallèles. Dans la pratique, tous les traitements ne bénéficient pas des mêmes améliorations quand ils sont effectués en parallèle : en effet, si beaucoup plus de communications que de calculs sont nécessaires, l'apport des ressources additionnelles peut devenir minime. Heureusement, certains programmes comportent des parties qui sont indépendantes les unes des autres et peuvent être exécutées simultanément. Actuellement, le concept de *meta-traitement* consiste à généraliser les mécanismes utilisés par les machines parallèles à des systèmes répartis moins fortement couplés (par exemple des réseaux longue distance). Un *serveur d'applications* a une tâche à résoudre, qui nécessite beaucoup de temps de calcul et peu de communications. Des ordinateurs distants téléchargent alors un programme depuis le serveur puis l'exécutent localement. Les résultats calculés localement par le client sont envoyés au serveur, qui les prend en compte lors des connexions suivantes. Au bout du compte, les serveurs se limitent à collecter les résultats calculés par des ordinateurs qui n'appartiennent pas forcément à l'organisme propriétaire du serveur, ce qui permet des économies substantielles.

4. **Simplifier :** La conception d'un système peut souvent être simplifiée en découpant celui-ci en modules, dont chacun implante une partie des fonctionnalités tout en communiquant avec les autres modules. Un système général peut alors être vu comme une *collection de processus coopérants*. Par exemple, un programme permettant simultanément à l'utilisateur d'imprimer une image, de recevoir et d'afficher des données vidéo, et de passer des ordres d'achat en bourse est invariablement très compliqué à écrire et à vérifier si on le programme de manière monolithique. A l'inverse, chaque tâche effectuée par le programme peut être codée et testée de manière indépendante, puis les trois

modules exécutés simultanément, par exemple en utilisant un système d'exploitation multi-processus.

5. **Fiabiliser** : Les systèmes répartis rendent possible le bon fonctionnement général du système même en cas de défaillance de certains composants. Typiquement, un tel comportement est obtenu en répliquant l'exécution d'applications sur différents processeurs, puis en filtrant les résultats obtenus par chacun d'entre eux pour établir le résultat correct. La fiabilisation des systèmes répartis constitue l'objectif principal des travaux menés dans le cadre de cette thèse, aussi la section 1.2 lui est-elle consacrée.

1.1.2 Spécificités

Pour réaliser les différentes fonctions demandées à un système réparti ont été développés des algorithmes spécifiques adaptés au contexte réparti : les *algorithmes répartis*. Les différences avec les systèmes centralisés tiennent en trois points essentiels :

1. **Localité des Informations** : Dans un système *centralisé*, l'*état global* du système peut être connu à tout instant par le programme en cours d'exécution : cet état global est en général déterminé seulement par l'état des variables et par la valeur du compteur de programme. Dans un système *réparti*, chaque programme en cours d'exécution (ou processus) ne possède qu'une *connaissance locale* donc partielle de l'état du système. Une hypothèse courante est de supposer qu'un processeur peut recevoir des informations des processeurs auxquels il est directement connecté (voir section 1.1.3), et de se baser sur ces informations pour exécuter un algorithme. Cependant, puisque transmettre des informations prend un certain temps, les informations reçues peuvent être obsolètes en raison du changement éventuel des variables gérées par le processeur émetteur depuis l'envoi des informations. D'autre part, l'état du système de communication ne peut jamais être observé directement par les processeurs. Les informations qui y sont relatives ne peuvent êtres que déduites des émissions et des réceptions de messages.
2. **Localité du Temps** : Dans un système centralisé, l'unique processeur exécute les actions de son programme séquentiellement, à une vitesse donnée. Il lui est facile de déterminer combien d'actions ont été nécessaires à l'établissement d'une tâche, combien de temps il a fallu pour que le système donne un résultat correct. Dans un système réparti, les événements ne sont plus totalement ordonnés mais organisés selon une relation d'ordre partiel. Pour certains de ces événements, il est possible de décider que l'un se produit avant l'autre (deux actions successives exécutées sur un même processeur, l'émission puis l'arrivée d'un message dans un canal de communication), mais pour beaucoup d'autres, aucune hypothèse ne peut être faite.
3. **Non Déterminisme :** Du fait de l'hétérogénéité possible des processeurs et des liens de communication dans un système réparti, il est tout à fait possible que ces différents composants agissent à des vitesses différentes, et par conséquent induisent des comportements non déterministes.

1.1.3 Hypothèses Pertinentes

Nous avons vu dans la section 1.1 que les systèmes répartis couvrent une large gamme de systèmes informatiques, qui va des machines multiprocesseurs aux réseaux d'ordinateurs internationaux (comme Internet). D'une manière générale, plus les communications sont rapides et plus le volume d'informations que l'on peut échanger est important. Cette quantité d'informations influe directement sur les hypothèses pertinentes dans différents systèmes répartis.

Les hypothèses sont habituellement classées de la plus restrictive vers la plus laxiste. Naturellement, un système réparti qui fonctionne correctement sous les hypothèses les plus laxistes supporte sans aucune modification des hypothèses plus restrictives. Au contraire, un système qui ne fonctionne que sous des hypothèses restrictives ne fonctionne pas dans un système plus laxiste sans l'adjonction d'un sous-système chargé de transformer les hypothèses (voir section 1.1.4 pour quelques exemples). Toutefois, de tels sous-systèmes n'existent pas toujours.

Communications Un système réparti est simplement défini par un ensemble de processeurs reliés par des liens de communication leur permettant d'échanger des informations. Cependant, aucune hypothèse n'est faite sur les capacités de chaque processeur à communiquer avec chacun des autres processeurs.

L'hypothèse la plus restrictive est celle de la communication *globale* : un lien de communication permet à tout processeur de communiquer avec tous les autres. Puis vient la communication *multipoints* : un lien de communication permet à un sous-ensemble des processeurs de communiquer entre eux. Enfin vient la communication *point à point* : un lien de communication permet à exactement deux processeurs de s'échanger des informations. Dans la suite de la thèse, nous utilisons la communication point à point, qui seule permet au système de s'adapter sans modifications aux réseaux à grande distance.

Dans le cas de la communication point à point, la *topologie* définit comment sont agencés les différents liens de communication par rapport aux processeurs. De manière traditionnelle, on représente les possibilités de communication par un graphe (le *graphe de communication*) où les processeurs sont les nœuds du graphe et les liens sont les arêtes (ou les arcs si les liens sont unidirectionnels). De nombreux algorithmes répartis utilisent des topologies particulières : la chaîne (figure 1.1.A), l'anneau (figure 1.1.B), l'étoile (figure 1.1.C), l'arbre (figure 1.1.D), la grille 4-connectée (figure 1.1.E), la grille 8-connectée (figure 1.1.F).

De manière encore plus restrictive, un lien de communication peut ne permettre les échanges d'informations que dans un seul sens. Par exemple, un lien l reliant deux processeurs P_1 et P_2 permet à P_1 d'envoyer des informations à P_2 mais pas à P_2 d'envoyer des informations à P_1. Ce type de communication est justifié par exemple dans le cas des réseaux par satellite, où la puissance des émetteurs et leur localisation influent sur leurs capacités de communication. Un lien de communication point à point où les échanges ne se font que dans un seul sens est un lien *unidirectionnel*.

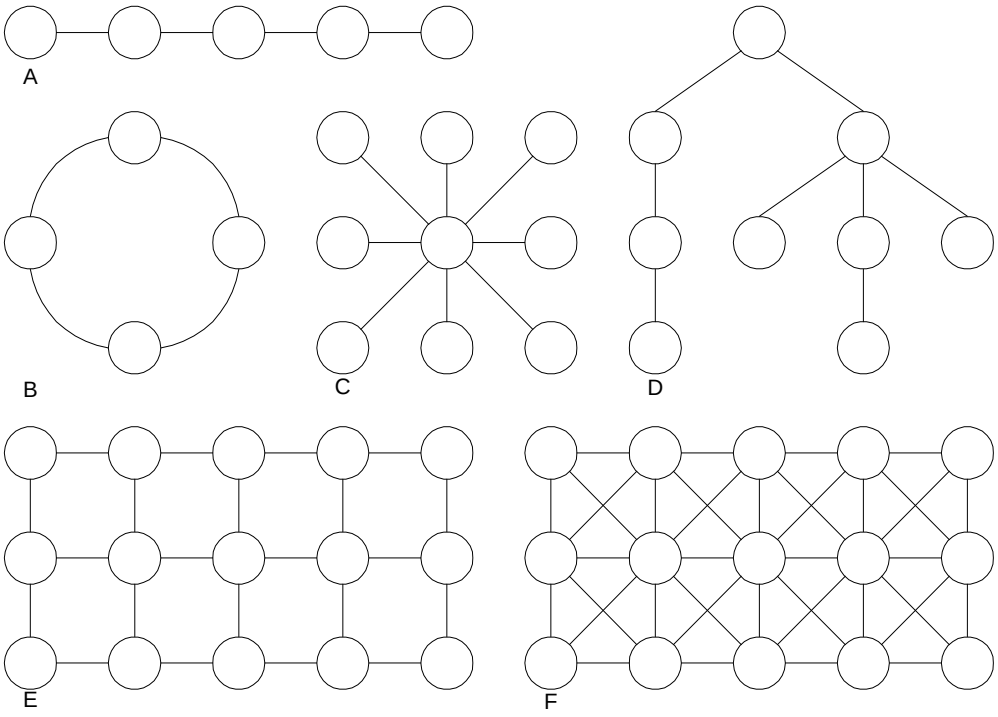

FIG. 1.1 – Quelques Topologies non-Orientées Usuelles.

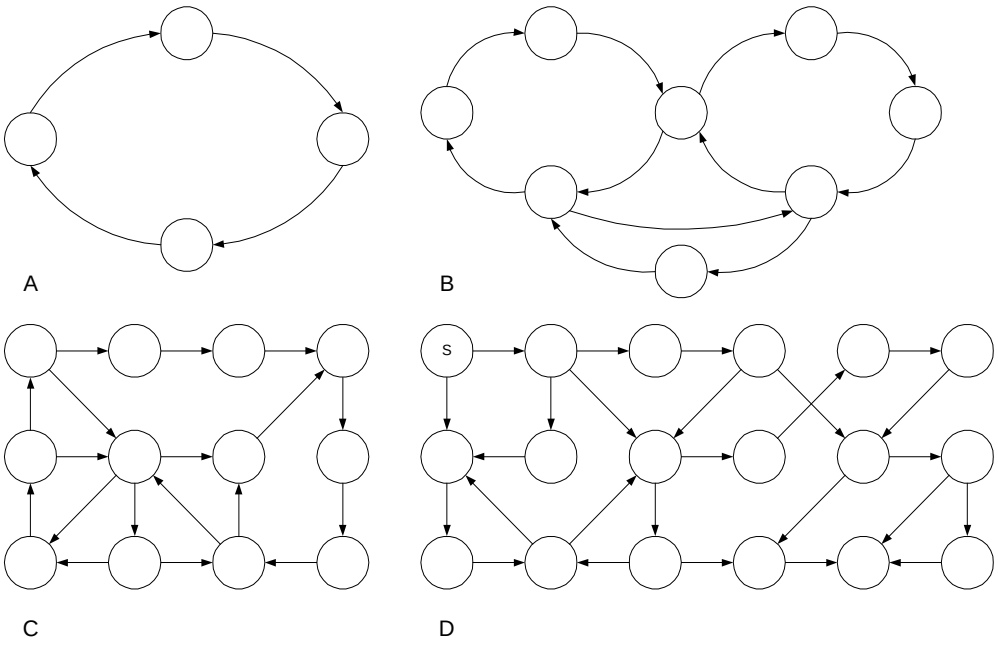

FIG. 1.2 – Quelques Topologies Orientées Usuelles.

De manière similaire aux topologies usuelles bidirectionnelles, il existe des topologies de référence lorsque les liens de communications sont unidirectionnels. Parmi celles-ci, on peut citer, du plus restrictif au plus laxiste : l'anneau unidirectionnel (chaque processeur possède un unique ascendant et un unique descendant, voir figure 1.2.A), le graphe eulérien (chaque processeur possède autant d'ascendant que de descendants, voir figure 1.2.B), le graphe fortement connexe (chaque processeur peut transmettre au moins indirectement des informations à chaque processeur du réseau, voir figure 1.2.C), le graphe avec source (au moins un processeur peut transmettre au moins indirectement des informations à chaque processeur du réseau, voir figure 1.2.D). Notons que chaque catégorie de graphes englobe la précédente : ainsi un algorithme réparti qui fonctionne dans la catégorie la plus générale (graphe avec source) fonctionne également dans un réseau fortement connexe, eulérien ou en anneau unidirectionnel.

Dans le contexte particulier de la tolérance aux défaillances (voir section 1.2), l'unidirectionnalité des liens de communications ne permet pas d'utiliser la technique classique de l'acquittement sur réception de message, qui garantit à l'émetteur du message que celui-ci a bien été reçu. En effet, un lien bidirectionnel (qui permet la communication dans les deux sens) peut être considéré comme l'ensemble de deux liens unidirectionnels liant les mêmes processeurs et dont les sens sont différents (voir figure 1.3).

Synchronisation Plusieurs hypothèses peuvent être faites concernant les vitesses relatives des composants du système : tous les composants fonctionnent à la même vitesse (système

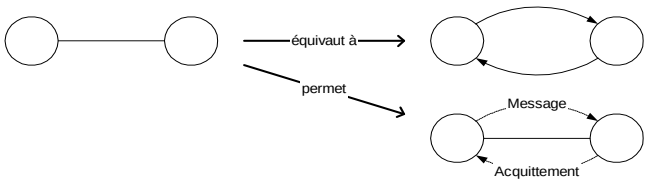

FIG. 1.3 – La Bidirectionnalité des liens permet l'acquittement des informations.

synchrone), il existe une borne connue entre la vitesse du composant le plus rapide et celle du composant le moins rapide (système *partiellement synchrone*), il n'existe aucune borne entre la vitesse du composant le plus rapide et celle du composant le moins rapide (système *asynchrone*).

Connaissances Globales Une première hypothèse a trait aux identifiants des processeurs. Dans le réseau Internet, chaque ordinateur connecté dispose d'une adresse unique qui lui permet de se distinguer des autres ; un tel système réparti est appelé un *système avec identifiants*. Dans une machine parallèle, on a bien souvent un ou plusieurs processeurs distingués qui régulent le système et distribuent les tâches à effectuer sur les différents processeurs du système ; un tel système est *semi-uniforme*. Enfin, un système réparti où tous les processeurs sont indistinguables est *uniforme* ; dans la pratique, de nombreux problèmes sont insolubles dans les systèmes uniformes symétriques à moins d'utiliser des algorithmes *probabilistes*.

Une seconde hypothèse concerne les connaissances globales sur le système dont disposent les processeurs qui le constituent. L'hypothèse la plus forte est celle de *topologie* : chaque processeur du système peut alors construire toutes les données globales ayant trait au réseau dont il a besoin : borne sur la taille du réseau, taille du réseau, liste de ses voisins à distance n, liste des processeurs avec qui il est capable de communiquer, ou encore table de routage. Evidement, si un algorithme fonctionne correctement sans aucune de ces informations globales, il fonctionne aussi si on les lui fournit, et dans la plupart des cas il effectue son travail beaucoup plus rapidement.

1.1.4 Problèmes Classiques

La plupart des problèmes classiques dans le cadre réparti ont trait au relâchement des contraintes induites par les hypothèses présentées dans la section 1.1.3. Par exemple, en construisant un anneau virtuel sur un réseau quelconque, il est possible d'exécuter un algorithme qui requiert une topologie en anneau dans un système où le graphe de communication est en fait un arbre.

Construction de Topologies Virtuelles Si certains algorithmes répartis sont conçus pour être exécutés sur des topologies bien connues, c'est principalement parce que ces to-

pologies facilitent le contrôle de l'exécution du programme. Il arrive fréquemment que les systèmes réels sur lesquels sont implantés les algorithmes répartis ne possèdent pas la topologie requise. Il est alors nécessaire de faire appel à un autre algorithme réparti qui construit virtuellement la topologie attendue par l'algorithme de plus haut niveau. Ainsi, de nombreux algorithmes répartis existent pour la construction d'arbres ou d'anneaux virtuels dans des réseaux plus généraux.

Construction d'Informations Globales Cette classe d'algorithmes consiste à échanger des informations locales de manière à construire de manière répartie une information globale. Des problèmes classiques sont la mise à jour de topologie (chaque processeur possède une connaissance complète de la topologie du graphe de communication du système), le routage (chaque processeur connaît un moyen d'atteindre directement ou non tout autre processeur du réseau), le recensement (chaque processeur connaît tous les autres processeurs présents sur le système), l'élection (un unique processeur est distingué sur le réseau), la taille (chaque processeur connaît le nombre de processeurs du système).

Diffusion d'Informations Quand un système réparti utilise des communications point à point, il est toutefois nécessaire de disposer de primitives de communications globales : la *diffusion* (un processeur transmet des informations à tous les autres), l'*échange total* (tous les processeurs transmettent des informations à tous les autres). Ces algorithmes rendent possible l'exploitation d'algorithmes prévus pour des systèmes à communications globales dans des systèmes à communication point à point ou multipoints. Pour plus de détails, le lecteur intéressé pourra se référer à [dR94].

Synchronisation Un algorithme réparti fonctionnant dans un système asynchrone est généralement difficile à concevoir, mais il fonctionne dans tous les autres systèmes synchrones ou partiellement synchrones, puisque ces deux derniers sont des cas particuliers des systèmes asynchrones. A l'inverse, un algorithme réparti fonctionnant dans un système synchrone est plus facile à concevoir, puisque le non-déterminisme du comportement peut être partiellement supprimé, mais pour qu'il fonctionne dans un système synchrone, il faut utiliser en parallèle un algorithme réparti spécial appelé *synchroniseur*, qui simule des processeurs synchrones dans un système asynchrone. Généralement, cette transformation s'accompagne d'une perte de performances significative.

Exclusion Mutuelle Les processeurs du système ont accès à une ressource partagée mais qui ne peut être accédée que par un processeur à la fois. L'exclusion mutuelle garantit que l'accès à cette ressource est sérialisé et équitable : les processeurs utilisent la ressource partagée à tour de rôle et infiniment souvent.

1.2 Tolérance aux défaillances

L'augmentation du nombre des éléments constitutifs d'un système réparti contribue à l'augmentation de la probabilité que l'un de ces éléments tombe en panne durant l'exécution de l'algorithme. Pour des applications où l'intervention d'un opérateur est impossible, ou bien des programmes dont l'exécution est longue et non vérifiable, il est essentiel de pouvoir pallier de manière logicielle aux déficiences matérielles. Plusieurs types de défaillances peuvent être répertoriés, suivant leur localisation (processeur ou lien) et leur nature.

Les *défaillances définitives* ont été les premières étudiées. Dans le cas d'un processeur, cela signifie que celui-ci stoppe définitivement son exécution ; dans le cas d'un lien, ce peut être traduit par une coupure définitive de la liaison de communication.

Les *défaillances intermittentes* sont plus difficiles à pallier. Un processeur peut avoir un comportement dit *byzantin*, où il exécute un autre algorithme que celui prévu ; quand un lien de communication peut manipuler les messages en transit (perte, duplication, déséquencement, modification).

Les *défaillances transitoires* sont similaires aux défaillances intermittentes mais suffisamment rares pour qu'elles puissent être supposées ne plus se produire après un instant donné. Pour un processeur, de telles défaillances peuvent se produire suite à une panne immédiatement suivie d'une réparation, ou encore lors de défaillances temporaires de l'alimentation électrique. Pour un lien de communication, des débranchements et branchements successifs, ainsi que des perturbations électromagnétiques ponctuelles, peuvent conduire à des problèmes similaires.

Pour corriger ces différents types de problèmes, deux approches complémentaires ont été développées : *la tolérance aux pannes* et l'*auto-stabilisation*.

1.2.1 La tolérance aux pannes

Lorsque certains des composants du système sont en panne la plupart du temps (c'est le cas lors des défaillances définitives ou intermittentes), on s'attache à garantir quoi qu'il arrive un fonctionnement correct des éléments non fautifs. Cette approche, classiquement appelée tolérance aux pannes, consiste à faire effectuer par chaque composant de base une série de vérifications avant d'exécuter chaque instruction de l'algorithme réparti, de manière à assurer à chaque instant une conformité à la spécification.

Le plus souvent, pour que le problème considéré admette une solution, on se trouve dans l'obligation de faire des hypothèses limitant le nombre de pannes pouvant survenir au cours d'une exécution. Ainsi de nombreux résultats d'impossibilité ont été mis en évidence, notamment dans le cas des systèmes répartis asynchrones, et même dans le cas où les fautes sont limitées aux processeurs (les liens de communication sont supposés fiables). Historiquement, le problème du consensus (tous les processeurs non fautifs doivent se mettre d'accord sur une valeur unique) a fait l'objet du plus grand nombre de recherches. Un résultat fondamental ([FLP85]) montre que le problème de consensus est insoluble de manière déterministe dans un système asynchrone même si la défaillance est définitive et limitée à un seul processeur.

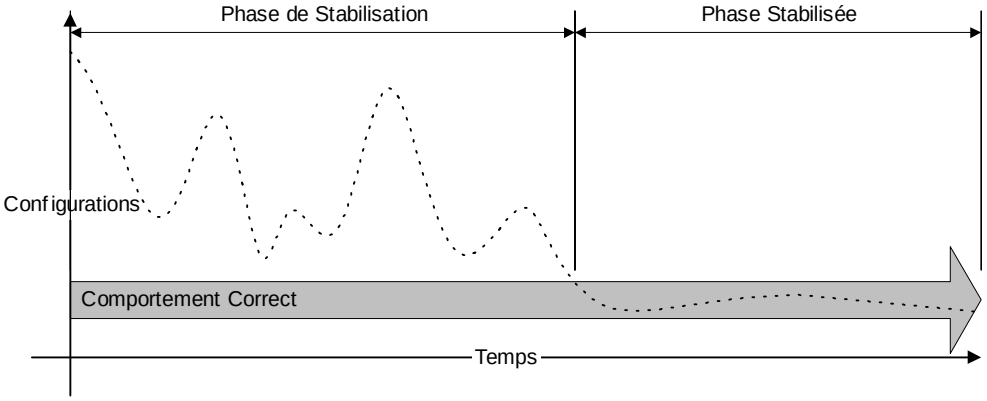

FIG. 1.4 – L'Auto-stabilisation

Dans le cas des défaillances intermittentes des liens de communications, les recherches ont surtout porté sur la construction de liens fiables (pour des messages de haut niveau) à partir de liens non fiables (pour des messages de bas niveau). Si le cas de la perte équitable ne pose pas problème ([Ste76, BSW69]), cette construction est impossible dans le cas où les liens d'origine peuvent dupliquer (de manière finie) et déséquencer des messages ([WZ89]).

1.2.2 L'auto-stabilisation.

A l'inverse, lorsque tous les composants du système fonctionnent correctement la plupart du temps (c'est le cas lors des défaillances transitoires), on peut accepter une interruption temporaire de service lors d'une corruption générale du système, mais néanmoins garantir un retour à la normale au bout d'un temps fini. Cette approche, l'auto-stabilisation, consiste à toujours agir comme si tous les autres composants du système étaient corrects. Évidemment, en cas de panne, le système peut ne pas être conforme à sa spécification (*phase de stabilisation*), mais si l'algorithme est auto-stabilisant, alors après un temps fini (généralement borné) et en l'absence de nouvelles pannes, le système est de nouveau conforme à sa spécification (*phase stabilisée*).

A la différence de la tolérance aux pannes, l'auto-stabilisation ne fait aucune restriction sur la portion du système qui peut être atteinte par des défaillances. Les algorithmes auto-stabilisants partent du postulat que les défaillances transitoires ne peuvent agir que sur la partie volatile des composants du système : la mémoire vive des processeurs et les messages en transit dans les liens de communication. La notion d'auto-stabilisation a été introduite en 1974 par Dijkstra ([Dij74]), mais sa portée dans le domaine de la résistance aux défaillances n'a été soulignée par Lamport que dans [Lam94]. La recherche en auto-stabilisation a principalement porté sur la conception d'algorithmes auto-stabilisants ([IJ90, AKY90, BCD95, TB96]), ainsi que sur la transformation automatique d'algorithmes classiques en leurs équivalents auto-stabilisants ([KP93, BD93]).

En plus de la tolérance aux pannes transitoires, les algorithmes auto-stabilisants présentent des avantages :

1. Aucune initialisation particulière n'est requise pour assurer le bon fonctionnement de l'algorithme. En effet, même si chacun des processeurs démarre son programme dans un état quelconque, le système fini par exhiber un comportement correct.
2. L'algorithme fonctionne de la même manière pour des topologies de réseau évoluant dynamiquement. Un algorithme auto-stabilisant qui calcule une fonction dépendante de la topologie (calcul de tables de routage ou d'un arbre couvrant sur un réseau) converge vers une nouvelle solution après que la topologie du réseau ait été modifiée.

Par contre, plusieurs problèmes surviennent lorsqu'on utilise des algorithmes auto-stabilisants :

1. A l'origine, l'exécution de l'algorithme ne correspond pas nécessairement à une exécution correcte (par exemple, plusieurs jetons peuvent être en circulation dans un réseau *Token Ring*).
2. Les performances peuvent être inférieures à celles de leurs équivalents non stabilisants lorsqu'il n'y a pas de défaillances transitoires.
3. Il n'y a pas de détection de la terminaison de l'algorithme. Comme il n'est pas possible de s'apercevoir qu'une configuration légitime a été atteinte, les éléments constitutifs du système ne sont jamais avertis du fait que leur fonctionnement est devenu fiable.

Le premier de ces aspects oblige le système à être capable de supporter la non conformation aux spécifications pendant un certain temps. Le second aspect est crucial lors de l'application d'algorithmes auto-stabilisants à des protocoles industriels.

1.2.3 Approches Multi-tolérantes

Quand le système réparti considéré est sujet à plusieurs types de défaillances, il est évidemment souhaitable de pouvoir les gérer.

Défaillance de Processeurs et Communications non Fiables Récemment, [BCBT96] et [DF97] se sont intéressés au cas de systèmes contenant non seulement des processeurs défaillants, mais aussi des liens de communication pouvant perdre des messages.

Auto-stabilisation et Défaillances de Processeurs On souhaite tolérer des défaillances transitoires sur la totalité du système (le système doit être auto-stabilisant) et des défaillances permanentes (voire intermittentes) sur les processeurs.

Une première approche est de considérer des systèmes où les pannes de processeurs sont toujours *permanentes*. On tolère des défaillances transitoires sur la totalité du système et des défaillances définitives sur un nombre limité de processeurs ([AH93, BKM97]).

Une deuxième approche est de considérer des systèmes où les pannes de processeurs sont *intermittentes* (c'est le cas des processeurs byzantins). On tolère des défaillances transitoires

sur la totalité du système et des défaillances intermittentes sur un nombre limité de processeurs (par exemple, voir [DW95]).

Auto-stabilisation et Communications non Fiables On souhaite tolérer des défaillances transitoires sur la totalité du sytème (le système doit être auto-stabilisant) et des défaillances permanentes (voire intermittentes) sur les liens de communication. Plusieurs solutions à ce problème ont été présentées, en particulier dans [AB93, KP93].

Certaines pannes de liens peuvent être assimilées à des changements de topologie : un lien qui tombe en panne est considéré comme coupé ; un lien qui était en panne et qui se trouve réparé est considéré comme un nouveau lien apparaissant spontanément. Dans les deux cas, les processeurs qui sont attachés au lien sont avertis que ce lien n'existe plus ou vient d'apparaître. Les protocoles *super-stabilisants* ([DH97]) sont auto-stabilisants d'une part, et capables de supporter des changements de topologie limités (un seul lien peut apparaître ou disparaître à la fois). Les protocoles *instantanément stabilisants* ([JADT99, BDPV99]) sont auto-stabilisants mais leur temps de stabilisation est égal à zéro ; il sont donc en mesure de supporter un nombre arbitraire de changement de topologie au cours de leur exécution. Les algorithmes 3 et 4 présentés au chapitre 5 sont instantanément stabilisants pour leurs spécifications respectives. Il convient toutefois de noter que tous les algorithmes instantanément stabilisants connus nécessitent une topologie particulière (par exemple un arbre) : si les changements de topologies induits par les ajouts ou les suppressions de liens de communications modifient la nature de cette topologie (par exemple en ajoutant un ou des cycles dans un réseau arborescent), alors le système risque de ne plus stabiliser.

Dans [DT98a], l'intermittence des défaillances des liens de communication est *stricte*. Les liens perdent *équitablement* des messages, les dupliquent *de manière finie*, les déséquencent sans restriction.

1.3 Critères d'efficacité

L'efficacité en espace peut être facilement décrite au moyen de l'espace mémoire requis par l'algorithme réparti utilisé, à la fois sur chaque processeur et sur chaque message envoyé.

Dans la littérature, l'efficacité en temps des algorithmes auto-stabilisants est mesurée par le temps de stabilisation dans le pire des cas du système présenté. Ce critère, s'il est important, n'est néanmoins significatif qu'en présence de défaillances transitoires.

Dans cette thèse, nous nous intéressons, en plus des deux critères précédents, au *surcoût* de l'algorithme auto-stabilisant par rapport à un autre algorithme *non-stabilisant* dans le cas où le système considéré *ne subit pas de défaillances transitoires*. Pour établir des critères d'efficacité relatifs au surcoût, nous distinguons plusieurs types de problèmes répartis :

1. **Les problèmes statiques** : les différents processeurs présents sur le réseau collaborent à l'établissement d'un calcul global. Par exemple, la taille du réseau, l'élection d'un chef, la coloration d'un graphe, la construction d'un arbre couvrant ou d'un anneau virtuel. Ces problèmes ont la particularité qu'à partir d'un certain point de l'exécution,

les communications entre processeurs voisins sont figées. L'efficacité en temps pour ce type de problèmes a trait au temps de calcul nécessaire pour obtenir un résultat correct.

2. **Les problèmes dynamiques** : les différents processeurs présents sur le réseau collaborent à l'établissement d'un service pour les utilisateurs ou des programmes de plus haut niveau. Par exemple, l'exclusion mutuelle, la transmission de messages, la circulation de jeton. Ces problèmes ont la particularité que toutes les exécutions du système sont infinies. L'efficacité en temps pour ce type de problèmes a trait à la qualité du service fourni : combien de temps s'écoule-t-il entre deux passages du jeton ? Quel débit puis-je obtenir de mon protocole de communication ?

1.4 Résumé

Dans ce chapitre, nous avons présenté les système répartis comme comme un ensemble de processeurs communicant au moyen de liens de communication. Nous avons mis en évidence les hypothèses pertinentes lors de la conception de systèmes : topologie du graphe de communication, vitesses relatives des processeurs, connaissances globales disponibles. Nous avons également donné une typologie des fautes pouvant survenir dans les systèmes répartis et présenté l'auto-stabilisation, une technique permettant de pallier certaines défaillances transitoires.

Chapitre 2

Modèle

Dans ce chapitre, nous formalisons certains des aspects développés au chapitre 1. En particulier, nous donnons une définition formelle des systèmes répartis que nous considérons, à partir des éléments de base qui les constituent (les processeurs et les liens de communication). Nous développons plusieurs aspects ayant trait au comportement des systèmes répartis.

2.1 Références

De nombreux ouvrages traitent directement ou indirectement des systèmes répartis. Suivant la culture qui s'y rapporte, le modèle sous-jacent est différent, mais partage quelques caractéristiques communes. En particulier, la description d'un système réparti par un graphe ou les sommets représentent des processeurs (représentés sous forme de systèmes de transitions dans [Tel94], sous forme d'automates à entrées/sorties dans [Lyn96][1]) et les arêtes des liens de communication semble communément admise.

Des modèles de systèmes répartis spécialement adaptés à l'auto-stabilisation ont été développés lors de travaux antérieurs : en particulier, [Her91] utilise un système de transitions où l'atomicité est fine, [Pet98] un système de transition où l'atomicité est composée (le modèle à états). Plus spécifiquement, le modèle [Del95] permet de prendre en compte des algorithmes auto-stabilisants probabilistes et celui de [KM98] des algorithmes auto-stabilisants et tolérant des défaillances permanentes de processeurs.

Le modèle que nous avons choisi est globalement similaire à celui décrit dans [AW98]. C'est un système de transitions à atomicité fine, où le graphe de communication sous-jacent est explicité afin de prendre en compte des communications unidirectionnelles ou à destinataires multiples. Nous donnons dans la section 2.4.4 des critères permettant de hiérarchiser la notion d'atomicité des actions.

[1] D'autres approches peuvent être trouvées dans [RH90, JàJ92, Bar96].

2.2 Préliminaires

Dans cette section, nous répertorions les notations utilisées tout au long de la thèse concernant les notions classiques de graphes (dans la section 2.2.1) et de produit synchronisé d'automates (dans la section 2.2.2). En effet, les systèmes répartis que nous considérons sont constitués d'un ensemble de processeurs liés entre eux par des liens de communications, et il est plus facile, plutôt que de décrire directement le système tout entier, de définir celui-ci à partir des composants de base (processeurs et liens) synchronisés entre eux (caractéristique d'une interaction entre les processeurs). Par ailleurs, plusieurs chapitres utilisent des graphes pour représenter les capacités de communications entre les processeurs d'un système réparti ; le graphe est alors appelé le graphe de communication.

2.2.1 Graphes

Définition 2.1 (Graphe) *Un graphe $\mathcal{G} = (V, E)$ est* orienté *si et seulement si V est un ensemble non vide de nœuds et E est un ensemble de couples ordonnés distincts (u, v) de nœuds, appelées* arcs*, où u est l'origine et v la destination. Un graphe $\mathcal{G} = (V, E)$ est* non-orienté *si et seulement si V est un ensemble non vide de nœuds et E est un ensemble de paires distinctes (u, v) de nœuds distincts, apellées* arêtes*.*

Le *degré entrant* d'un nœud $v \in \mathcal{G}$, dénoté par $\delta^-(v)$ est égal au nombre de nœuds u tels que l'arc (u, v) est dans $\mathcal{G}$. Les arcs entrants de chaque nœuds v de $\mathcal{G}$ sont numérotés de 1 à $\delta^-(v)$. Nous dénotons par $Ind(u)$ l'indice de l'arc entrant (u, v) de v. De manière similaire, le degré sortant d'un nœud $v \in \mathcal{G}$, dénoté par $\delta^+(v)$ est égal au nombre de nœuds u tels que l'arc (v, u) est dans $\mathcal{G}$.

Dans le cas où $\mathcal{G}$ n'est pas orienté, le *degré* d'un nœud v de $\mathcal{G}$, dénoté par $\delta(v)$ est égal au nombre de nœuds u tels que l'arête (u, v) est dans $\mathcal{G}$.

Deux processeurs connectés par un arc ou une arête sont des *voisins*.

Définition 2.2 (Chemin) *Un* chemin *d'un nœud v_0 à un nœud v_k dans un graphe $\mathcal{G}$ est une liste d'arcs ou d'arêtes consécutifs de $\mathcal{G}$, $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$.*

La *longueur* de ce chemin est k. Si chaque v_i est unique, le chemin est *élémentaire*. Un *cycle* est un chemin où $v_0 = v_k$. Notons que dans un graphe non-orienté, un cycle est toujours de longueur au moins 3. Un graphe orienté sans cycle est appelé un *DAG* (de l'anglais *directed acyclic graph*).

Définition 2.3 (Forte Connexité) *La* composante fortement connexe *d'un nœud v de $\mathcal{G}$ est l'ensemble de tous les nœuds w de $\mathcal{G}$ tels qu'il existe un chemin orienté de v à u* et *un chemin orienté de u à v.*

Un graphe orienté $\mathcal{G}$ est *fortement connexe* si tous les nœuds de $\mathcal{G}$ possèdent la même composante fortement connexe.

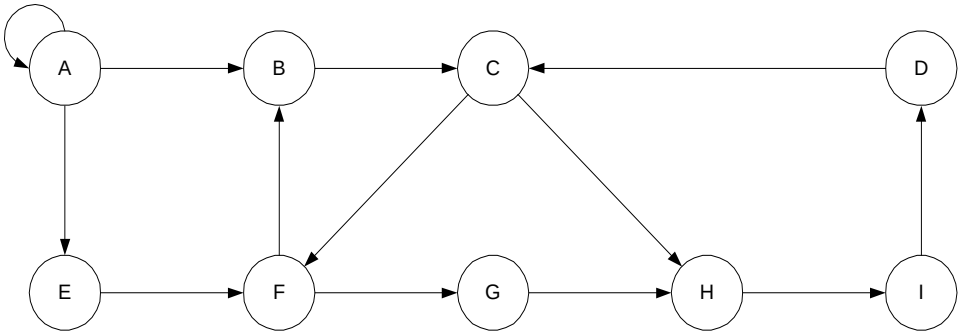

FIG. 2.1 – Un Graphe Orienté

Exemple La figure 2.1 présente un graphe orienté $\mathcal{G}$ comprenant 9 nœuds et 12 arcs. Le degré entrant du nœud F est 2 car seuls les arcs (E, F) et (C, F) sont dans l'ensemble des arcs de $\mathcal{G}$. Les nœuds H et G sont des voisins. Le chemin $c = (B, C), (C, F), (F, B)$ est un cycle car l'origine du premier arc de c et la destination du dernier arc de c sont identiques. La longueur de c est 3. Le nœud A possède un *arc bouclant*. Le graphe $\mathcal{G}$ n'est pas fortement connexe car A et B n'ont pas la même composante fortement connexe. En effet, la composante fortement connexe de A est $\{A\}$ alors que la composante fortement connexe de B est $\{B, C, D, F, G, H, I\}$.

Définition 2.4 (Distance) *Soient u et v deux nœuds de $\mathcal{G}$, Dist est une fonction à valeurs dans $\mathbb{N}$ et telle que $Dist(u, v)$ renvoie le nombre d'arcs sur un plus court chemin de u à v (si un tel chemin existe).*

Notons que dans un graphe non orienté connexe, la distance entre deux nœuds distincts est toujours définie.

Soit $MaxDist(u)$ la valeur maximale des $Dist(u, v)$ sur tous les nœuds v de $\mathcal{G}$. Le *diamètre* de $\mathcal{G}$ est la valeur maximale de $MaxDist(v)$ pour tous les nœuds v de $\mathcal{G}$. Un nœud u est un *centre* de $\mathcal{G}$ s'il n'existe aucun nœud v tel que $MaxDist(u) > MaxDist(v)$. Le *rayon* de $\mathcal{G}$ est la valeur $MaxDist(u)$ pour un centre u de $\mathcal{G}$.

Définition 2.5 (Descendants et Ascendants) *Un* descendant *(respectivement* ascendant*) d'un nœud v de $\mathcal{G}$ est un nœud u de $\mathcal{G}$ tel qu'il existe un chemin de v à u (respectivement de u à v) dans $\mathcal{G}$.*

L'ensemble des descendants de v dans $\mathcal{G}$ est dénoté par $\Gamma^+_{\mathcal{G}}(v)$, et $\Gamma^{+k}_{\mathcal{G}}(v)$ dénote l'ensemble des descendants u de v dans $\mathcal{G}$ tels que $Dist(v, u) = k$. L'ensemble des nœuds de $\Gamma^{+1}_{\mathcal{G}}(v)$ est celui des *descendants directs* de v dans $\mathcal{G}$. Enfin, un nœud v est un *puit* de $\mathcal{G}$ si et seulement si il n'a pas de descendants (*i.e.* $\Gamma^+_{\mathcal{G}}(v) = 0$).

De manière similaire, l'ensemble des ascendants de v dans $\mathcal{G}$ est dénoté par $\Gamma^-_{\mathcal{G}}(v)$, et $\Gamma^{-k}_{\mathcal{G}}(v)$ dénote l'ensemble des ascendants u de v dans $\mathcal{G}$ tels que $Dist(u, v) = k$. L'ensemble

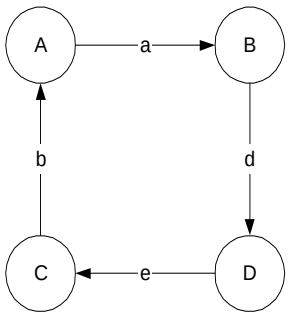

FIG. 2.2 – Automate A_1

des nœuds de $\Gamma_{\mathcal{G}}^{-1}(v)$ est celui des *ascendants directs* de v dans $\mathcal{G}$. Enfin, un nœud v est une *source* de $\mathcal{G}$ si et seulement si il n'a pas d'ascendants (*i.e.* $\Gamma_{\mathcal{G}}^{-}(v) = 0$).

Dans un graphe non-orienté, $\Gamma_{\mathcal{G}}^{k}(v)$ dénote l'ensemble des nœuds qui sont à distance k du nœud v dans le graphe $\mathcal{G}$. En particulier, $\Gamma_{\mathcal{G}}^{1}(v)$ est l'ensemble des *voisins* de v dans $\mathcal{G}$.

Un nœud v est *principal* dans $\mathcal{G}$ si et seulement si il est l'ascendant de tous les autre nœuds (*i.e.* tout nœud u de $\mathcal{G}$ est tel que $v \in \Gamma_{\mathcal{G}}^{-}(u)$). Soit i un entier, $Boule(u, i)$ dénote l'ensemble B des nœuds b tels que $Dist(b, u) \leq i$ (on a donc $Boule(u, i) = \{u\} \cup \Gamma_{\mathcal{G}}^{-1}(u) \cup \cdots \cup \Gamma_{\mathcal{G}}^{-i}(u)$).

2.2.2 Automates et Produit Synchronisé

Plutôt que de décrire un système réparti globalement, ce qui est à la fois fastidieux et difficile, on s'intéresse à représenter des composants de base sous la forme d'automates, qui sont des outils pratiques pour décrire graphiquement des machines informatiques.

Définition 2.6 (Automate) *Un* automate *est un triplet $A = (Q, E, T)$ où Q est un ensemble d'états, E est un ensemble d'actions, et $T \subset Q \times E \times Q$ est un ensemble de transitions.*

Dans la suite, si A est un automate, alors $Etats(A)$ dénote Q et $Actions(A)$ dénote E. Dans la littérature, E est souvent référencé comme un ensemble de *mots*. Une *phrase* d'un automate A est une séquence, finie ou infinie, de transitions (q_i, e_i, q_i') de A telles que $q_i' = q_{i+1}$ pour tout i. Le *langage* reconnu par l'automate A est alors l'ensemble des phrases de A. Dans la suite, la répétition d'une séquence d'actions $a_1 \dots a_n$ est notée $(a_1 \dots a_n)^{\omega}$.

Exemple Un automate peut être vu comme un graphe orienté où les arcs sont étiquetés par des actions. La figure 2.2 présente un automate $A_1 = (Q_1, E_1, T_1)$ où $Q_1 = \{A, B, C, D\}$, $E_1 = \{a, b, d, e\}$ et $T_1 = \{(A, a, B), (B, d, D), (D, e, C), (C, b, A)\}$. La figure 2.3 présente un automate $A_2 = (Q_2, E_2, T_2)$ où $Q_2 = \{A, B, C\}$, $E_2 = \{a, b, c\}$ et $T_2 = \{(A, a, B), (B, b, C), (C, c, A)\}$.

On introduit le concept d'action vide (notée $-$) qui signifie que l'automate ne fait rien. Chaque transition associée à l'action vide est du type $(C, -, C)$, c'est à dire que la configuration de départ et celle d'arrivée sont les mêmes pour une action vide.

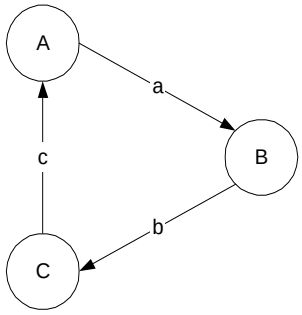

FIG. 2.3 – Automate A_2

Quand un système peut être décomposé en un ensemble d'automates n'interagissant pas entre eux (le système global est une collection de machines séquentielles sans liens entre elles), l'automate global décrivant le comportement de l'ensemble des automates est simplement un produit cartésien de tous les automates constitutifs.

Définition 2.7 (Produit Cartésien) *Le* produit *d'un ensemble* $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ *de* n *automates , avec* $A_i = (Q_i, E_i, T_i)$ *pour tout* i*, est un automate* $P = (Q, E, T)$ *où :*

1. $Q = Q_1 \times Q_2 \times \cdots \times Q_n$,
2. $E = \prod_{1 \leq i \leq n}(E_i \cup \{-\})$, *et*
3. $T = \left\{ \begin{array}{l} ((q_1, \ldots, q_n), (e_1, \ldots, e_n), (q'_1, \ldots, q'_n))| \\ \forall i \in \{1, \ldots, n\}, (e_i = - \wedge q_i = q'_i) \vee (e_i \neq - \wedge (q_i, e_i, q'_i) \in T_i) \end{array} \right\}$.

Exemple Sur la figure 2.4, l'automate produit des automates A_1 et A_2 décrits précédement est représenté. Le nombre d'états de cet automate est le produit des nombres d'états des automates qui le composent (soit $3 \times 4 = 12$). Afin de simplifier la figure, les transitions où tous les automates de base effectuent des actions vides n'ont pas été représentées.

Les systèmes répartis qui présentent un intérêt sont ceux où les actions d'un automate de base peuvent avoir une incidence sur les actions d'un autre automate de base. Pour rendre compte de ces interactions, on peut restreindre les comportements de l'automate en n'autorisant seulement certaines actions composées (ou synchronisées). Par exemple, seules les actions où un seul des automates composants agit peuvent être possibles. On utilise alors un *ensemble de synchronisation* (c'est à dire un ensemble d'actions) qui définit la liste des actions autorisées pour l'automate produit considéré.

Définition 2.8 (Produit Synchronisé) *Le* produit synchronisé *d'un ensemble de* n *automates* $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$*, avec l'ensemble de synchronisation* $S \subset \prod_{1 \leq i \leq n}(E_i \cup \{-\})$*, est un automate* $P = (Q, E, T)$ *où :*

1. $Q = Q_1 \times Q_2 \times \cdots \times Q_n$,

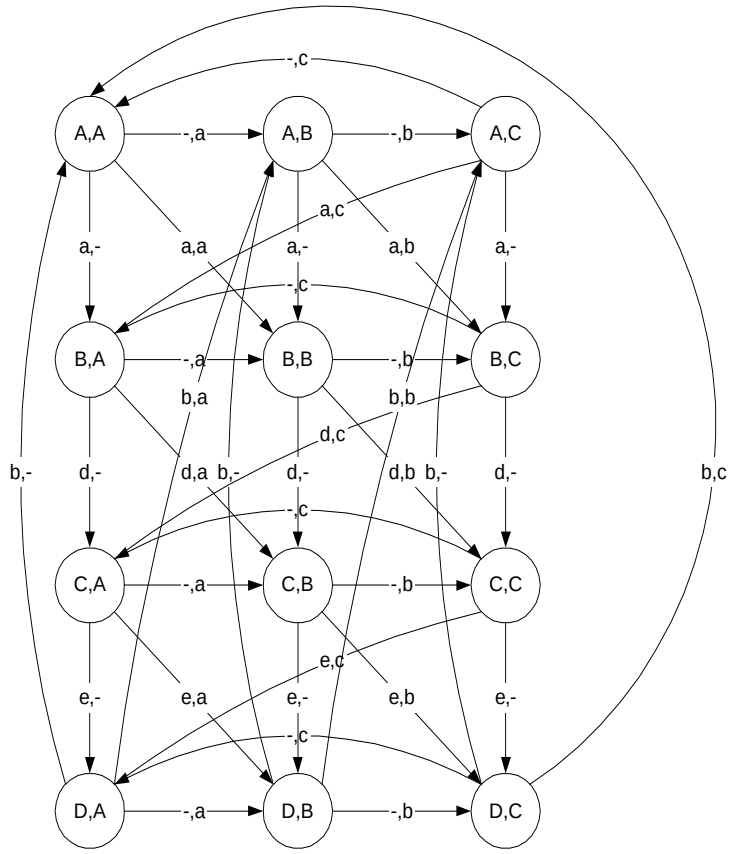

FIG. 2.4 – Produit des Automates A_1 et A_2

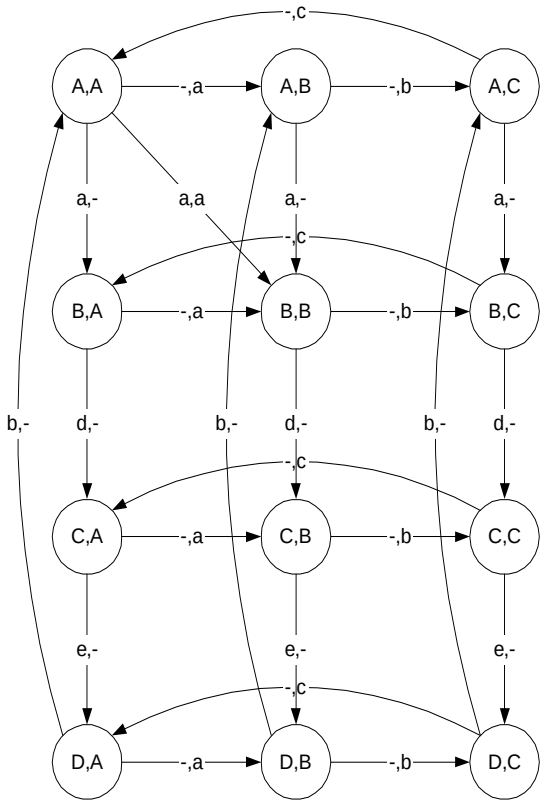

FIG. 2.5 – Produit Synchronisé des Automates A_1 et A_2

2. $E = S$, *et*

3. $T = \left\{ \begin{array}{l} ((q_1, \ldots, q_n), (e_1, \ldots, e_n), (q'_1, \ldots, q'_n))| \\ (e_1, \ldots, e_n) \in S, \\ \forall i \in \{1, \ldots, n\},\ (e_i = - \wedge q_i = q'_i) \vee (e_i \neq - \wedge (q_i, e_i, q'_i) \in T_i) \end{array} \right\}.$

Exemple Sur la figure 2.5, l'automate produit des automates A_1 et A_2 décrits précédement est représenté, avec l'ensemble de synchronisation

$$S = \{(a, a), (-, a), (-, b), (-, c), (a, -), (b, -), (d, -), (e, -)\}$$

Un tel ensemble de synchronisation impose que dans chaque action de l'automate produit, soit les deux automates A_1 et A_2 exécutent simultanément leur action a, soit un seul des automates composants effectue une action. Le nombre d'états de cet automate est toujours le produit des nombres d'états des automates qui le composent (soit $3 \times 4 = 12$). Cependant, le nombre de ses actions se trouve réduit par les actions incluses dans l'ensemble de synchronisation.

2.3 Eléments de Base

Les processeurs et les liens qui constituent les éléments de base d'un système réparti sont modélisés avec des automates, suivant des règles qui sont décrites ci-après. Intuitivement, dans un système réparti, les liens doivent permettre aux processeurs de communiquer entre eux. Certaines actions des processeurs et des liens doivent alors être synchronisées. Il existe une partition de l'étiquetage des fonctions de transitions en trois ensembles éventuellement vides :

1. L : ensemble des actions de lecture (une action de lecture sur un processeur est synchronisée avec une action *Lit* si le lien est un registre et une action *Reçoît* si le lien est un câble) ;
2. I : ensemble des actions internes (ces actions n'ont pas à être synchronisées avec des processeurs ou des liens) ;
3. E : ensemble des actions d'écriture (une action d'écriture sur un processeur est synchronisée avec une action *Ecrit* si le lien est un registre et une action *Envoie* si le lien est un câble).

2.3.1 Liens

Registre Un *registre* r est un automate dont les états sont les valeurs du registre et dont la fonction de transition est représentée par un graphe complet étiqueté par une famille d'actions du type : *Ecrit(r,v)* écrit la valeur v dans le registre r. Par ailleurs, chaque état v de l'automate possède deux arcs bouclants étiquetés respectivement par *Ecrit(r,v)* et *Lit(r,v)* (lit la valeur v dans le registre r).

Exemple La figure 2.6 présente l'automate d'un registre pouvant prendre trois valeurs : 1, 2, et 3.

Câble Un *câble* c est un automate dont les états sont des multi-ensembles de valeurs (appelés *messages*) et dont la fonction de transition est représentée par un graphe étiqueté par deux familles d'actions : *Envoie(c,m)* ajoute le message m au câble c ; *Reçoît(c,m)* retire le message m du câble c.

Exemple La figure 2.7 représente l'automate d'un câble de taille bornée (il peut contenir au plus deux messages simultanément) et spécifique aux messages m_1 et m_2.

La figure 2.8 représente l'automate d'un câble de même taille que celui de la figure 2.7, mais qui possède la propriété de respecter l'ordre dans lequel les messages ont été envoyés. En d'autres termes, si la suite de messages $m_1m_2m_1m_1m_2$ est envoyée au moyen de l'action *Envoie*, alors la suite de messages $m_1m_2m_1m_1m_2$ est reçue au moyen de l'action *Reçoît*. Un tel câble est appelé un câble *FIFO* (de l'anglais *first in, first out*).

Définition 2.9 (Lien) *Un* lien l *désigne indifféremment un registre ou un câble.*

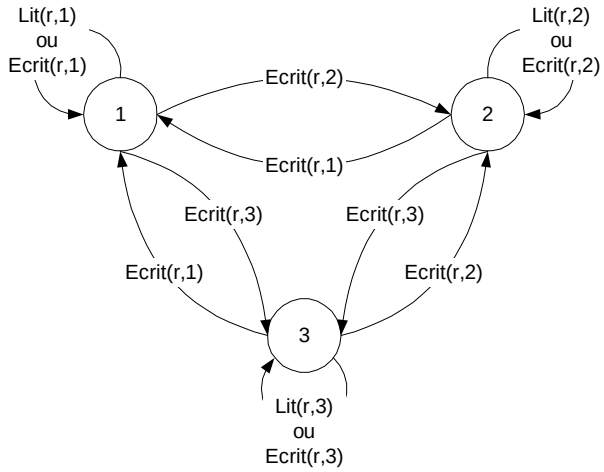

FIG. 2.6 – Un Registre

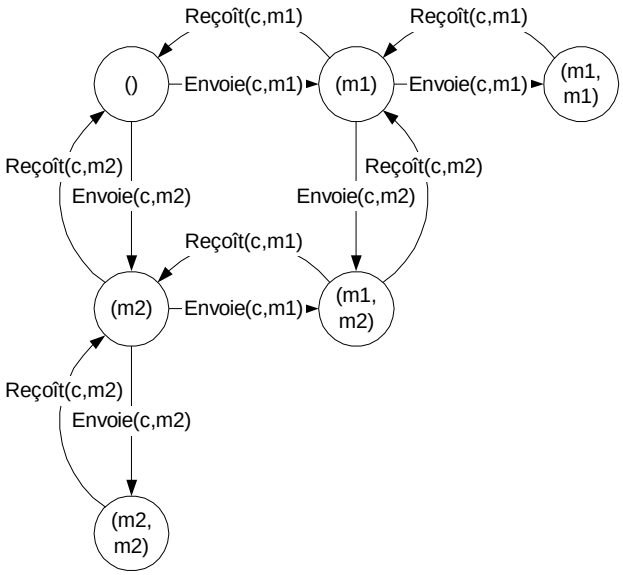

FIG. 2.7 – Un Câble

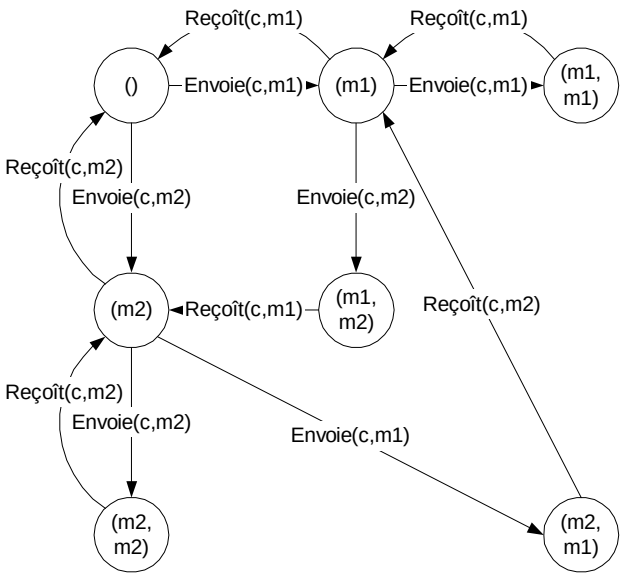

FIG. 2.8 – Un Câble FIFO

Même si les registres et les câbles sont répertoriés sous l'appellation générique de lien, il existe une différence fondamentale entre eux : dans un câble, il n'est pas toujours possible de lire des informations, et il n'est pas toujours possible d'écrire des informations. De plus, la lecture d'un message modifie l'état du câble, alors que ce n'est pas le cas d'un registre.

2.3.2 Processeurs

Les processeurs sont modélisés au sein du système réparti par des automates, mais décrits pour plus de lisibilité par des algorithmes locaux.

Définition 2.10 *Un* processeur p *est un automate spécifié par un algorithme.*

Algorithme L'algorithme d'un processeur est donné sous la forme d'une disjonction finie de *règles gardées*, également appelées *actions composées*. Les règles gardées sont constituées d'une *garde* et d'une *instruction* :

$\langle$Algorithme$\rangle \equiv \langle$Règle 1$\rangle|\langle$Règle 2$\rangle|\cdots|\langle$Règle $n\rangle$

$\langle$Règle$\rangle \equiv \langle$Garde$\rangle \rightarrow \langle$Instruction$\rangle$

La garde est une séquence (éventuellement vide) d'actions internes et de lecture (éléments de l'ensemble $I \cup L$) et conditionne l'exécution de l'instruction. L'instruction est une séquence non vide d'actions internes et d'écriture (éléments de l'ensemble $I \cup E$). Une règle dont la garde ne contient aucune action est *spontanée*. Notons qu'un processeur étant une machine *séquentielle*, il ne peut exécuter qu'une règle gardée à la fois. En particulier, il est incapable d'évaluer une garde s'il est en train d'exécuter l'instruction relative à une autre garde.

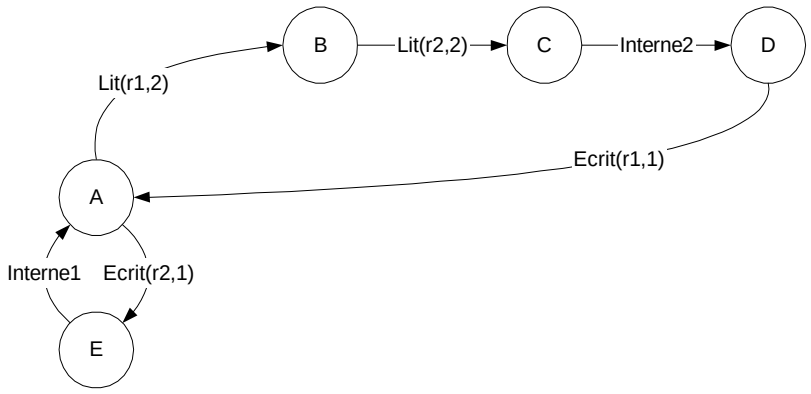

FIG. 2.9 – Un Processeur Exécutant les Actions Gardées A et B

Pour qu'un algorithme détermine un système de transitions modélisant un processeur, nous transformons son algorithme de la manière suivante :

1. Les variables de l'algorithme et son pointeur de programme déterminent les états de l'automate : chaque état représente une valeur possible du pointeur de programme et des variables de l'algorithme ;

2. Le code de l'algorithme détermine la fonction de transition de l'automate : chaque transition de l'automate est étiquetée par l'une des actions figurant dans l'une des règles gardées et chaque règle gardée induit une phrase dans l'automate.

Exemple Soit un processeur dont le code est donné sous la forme de deux règles gardées A et B, et qui suppose la possibilité de pouvoir communiquer au moyen de deux registres R_1 et R_2 :

$A \equiv \langle Lit(R_1, 2)\,;\, Lit(R_2, 2)\rangle \rightarrow \langle \text{Interne}_2\,;\, Ecrit(R_1, 1)\rangle$

$B \equiv \langle\rangle \rightarrow \langle \text{Interne}_1\,;\, Ecrit(R_2, 1)\rangle$

La garde de A contient deux opérations de lecture, l'instruction de A contient une action interne et une action d'écriture. La règle B est spontanée (sa garde ne contient aucune action), l'instruction de B contient une action interne et une action d'écriture.

La figure 2.9 présente un automate représentant le processeur dont le code est décrit par les deux règles A et B. Notons que comme le processeur n'utilise pas de variables, son état est donné par la valeur de son compteur de programme (la référence de la prochaine instruction à exécuter).

2.4 Systèmes Répartis

2.4.1 Définitions

Définition 2.11 (Système Réparti) *Un système réparti $\mathcal{S}$ est un couple (P, L), où P est un ensemble d'automates représentant des processeurs et L un ensemble d'automates représentant des liens.*

A partir d'un système réparti $\mathcal{S} = (P, L)$, il est possible de construire un graphe $\mathcal{G}$ modélisant la relation "est capable de transmettre des informations à". $\mathcal{G}$ est le *graphe de communication* de $\mathcal{S}$. Par exemple, si les liens de communications sont des registres partagés, alors l'arc (i, j) est présent dans $\mathcal{G}$ si et seulement si une action $Ecrit(R_{(i,j)}, x)$ fait partie de l'alphabet d'actions de $P_i \in P$ et une action $Lit(R_{(i,j)}, x)$ fait partie de l'alphabet d'actions de $P_j \in P$, et $R_{(i,j)}$ est un registre de L.

Définition 2.12 (Configuration) *Soit $\mathcal{S} = (P, L)$ un système réparti, avec $P = \{P_1, \ldots, P_n\}$ et $L = \{L_1, \ldots, L_{n'}\}$. Une configuration c de $\mathcal{S}$ est un vecteur des états de tous les automates du système (Formellement, $c \in (\text{Etats}(P_1) \times \cdots \times \text{Etats}(P_n) \times \text{Etats}(L_1) \times \cdots \times \text{Etats}(L_{n'}))$).*

L'ensemble des configurations de $\mathcal{S}$ est noté $\mathcal{C}$. Pour décrire le comportement d'un système réparti, il est possible de le représenter sous la forme d'un automate global, dont les états sont les configurations du système, et la fonction de transition la composition d'une ou plusieurs actions des automates du système. Cet automate global est obtenu par l'intermédiaire d'un *produit synchronisé* des automates de $\mathcal{S}$ suivant l'approche détaillée dans [Arn92, Del95], en utilisant un alphabet de synchronisation S tel que :

1. Exactement une action interne a lieu sur un automate et l'action vide $(-)$ a lieu sur les autres automates ;
2. Exactement deux actions de lecture identiques ont lieu sur deux automates (une sur un processeur et une sur un lien) et l'action vide a lieu sur les autres automates ;
3. Exactement deux actions d'écriture identiques ont lieu sur deux automates (une sur un processeur et une sur un lien) et l'action vide a lieu sur les autres automates.

Plus formellement, soit $\mathcal{S} = (P, L)$ un système réparti, avec $P = (p_1, \ldots, p_n)$ et $L = (l_1, \ldots, l_{n'})$. L'alphabet de synchronisation S du système est :

$$S = \left\{ \begin{array}{ll} (\underbrace{-, \ldots, -}_{k-1}, a_{p_k}, \underbrace{-, \ldots, -}_{n-k}, \underbrace{-, \ldots, -}_{k'-1}, a_{l_{k'}}, \underbrace{-, \ldots, -}_{n'-k'}) & a_{p_k} \text{ et } a_{l_{k'}} \text{ sont deux actions de même étiquette } (n \geq k \geq 1,\ n' \geq k' \geq 1) \\ (\underbrace{-, \ldots, -}_{l-1}, a_{p_l}, \underbrace{-, \ldots, -}_{l-k}, \underbrace{-, \ldots, -}_{n'}) & a_{p_l} \text{ est une action interne } (n \geq l \geq 1) \end{array} \right\}$$

Définition 2.13 (Action du Système) *Soit $\mathcal{S} = (P, L)$ un système réparti. Une* action *de $\mathcal{S}$ est un élément de son ensemble de synchronisation.*

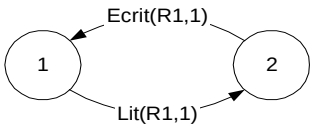

FIG. 2.10 – L'automate du processeur P_1

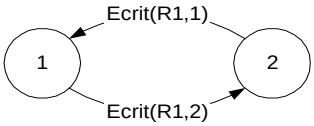

FIG. 2.11 – L'automate du processeur P_2

Exemple Considérons un système réparti constitué de deux processeurs P_1 et P_2 et d'un registre R_1. Le processeur P_1 dispose d'une unique action gardée $\langle Lit(R_1, 1)\rangle \rightarrow \langle Ecrit(R_1, 1)\rangle$, et le processeur P_2 d'une unique action spontanée $\langle\rangle \rightarrow \langle Ecrit(R_1, 2); Ecrit(R_1, 1)\rangle$. Comme aucun des processeurs P_1 et P_2 ne contient de variable locale, l'état est donné par la valeur du pointeur de programme de chaque processeur. Le registre R_1 peut prendre deux valeurs entières 1 et 2. Les automates correspondant aux processeurs P_1, P_2 et au registre R_1 sont donnés sur les figures 2.10, 2.11 et 2.12, respectivement.

Le produit synchronisé des processeurs P_1, P_2 et R_1 est donné figure 2.13 et utilise plusieurs conventions :

1. Les configurations du système sont des triplets $[a, b, c]$ où $a \in Etats(P_1)$, $b \in Etats(P_2)$ et $c \in Etats(R_1)$. Pour des questions de lisibilité de la figure, les crochets sont exclus de la représentation.
2. L'alphabet de synchronisation S est tel que :

$$S = \left\{ \begin{array}{ll} (Lit(R_1,1), -, Lit(R_1,1)), & \text{action de lecture de } R_1 \text{ par } P_1 \\ (Ecrit(R_1,1), -, Ecrit(R_1,1)), & \text{action d'écriture de } R_1 \text{ par } P_1 \\ \left[\begin{array}{l} (-, Ecrit(R_1,1), Ecrit(R_1,1)), \\ (-, Ecrit(R_1,2), Ecrit(R_1,2)) \end{array}\right] & \text{actions d'écriture de } R_1 \text{ par } P_2 \end{array} \right\}$$

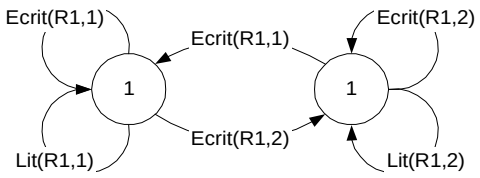

FIG. 2.12 – L'automate du registre R_1

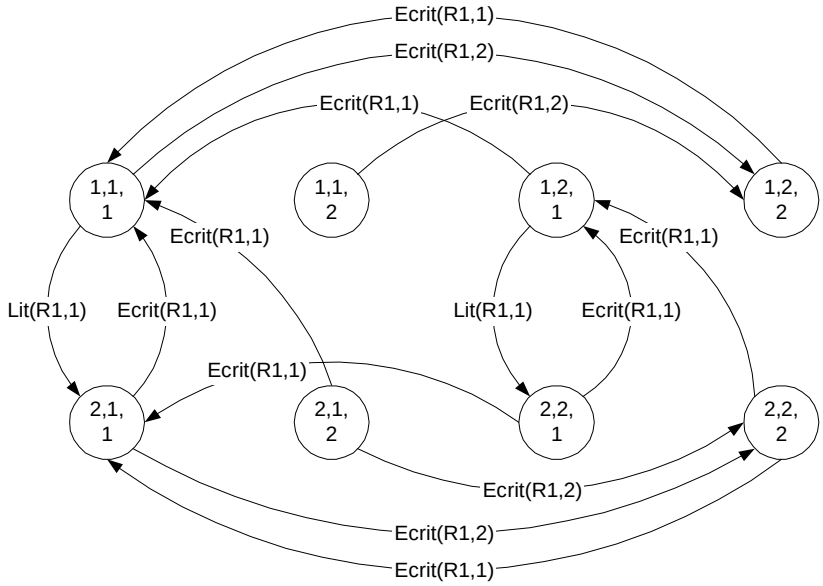

FIG. 2.13 – Le système réparti composé des processeurs P_1 et P_2 et du registre R_1

3. Pour des questions de lisibilité de la figure, chaque action de l'alphabet de synchronisation est remplacée par son action caractéristique. Par exemple, $(Lit(R_1, 1), -, Lit(R_1, 1))$ est remplacée par $Lit(R_1, 1)$.

Définition 2.14 (Exécution) *Soit $\mathcal{S}$ un système. Une* exécution *de $\mathcal{S}$, notée $e = c_1 a_1 c_2 a_2 \ldots$, est une séquence maximale de configurations et d'actions de $\mathcal{S}$ telle que c_{i+1} est atteinte depuis c_i par l'exécution de l'action a_i. La configuration c_1 est la* configuration initiale *de e.*

La séquence est maximale dans le sens où elle est soit infinie, soit finie mais aucune action de $\mathcal{S}$ n'est possible dans la dernière configuration (appelée configuration terminale). L'ensemble des exécutions de $\mathcal{S}$ est notée $\mathcal{E}$. L'ensemble des exécutions de $\mathcal{S}$ dont la configuration initiale est c est notée $\mathcal{E}_c$. L'ensemble des exécutions de $\mathcal{S}$ dont la configuration initiale fait partie d'un ensemble $C_1 \subset \mathcal{C}$ est notée $\mathcal{E}_{C_1}$.

2.4.2 Projections des Exécutions

Les *facteurs* permettent de définir de manière commode les propriétés portant sur une portion d'une exécution. De manière similaire, les *journaux* permettent de décrire les propriétés portant sur les configurations (et aussi à définir les *spécifications* des systèmes répartis, voir section 2.4.5), et les *traces* permettent de décrire des propriétés portant sur les actions (et en particulier les propriétés liées à l'*ordre* dans lequel les actions apparaîssent, qui se rapporte à la notion de *démon*, voir section 2.4.4).

Définition 2.15 (Facteur) *Un* facteur *d'une exécution $e \in \mathcal{E}$ est une sous-séquence de e, dont le premier terme (respectivement le dernier terme si le facteur est fini) est une configuration.*

Par exemple, $f = c_2a_2c_3$ est un facteur fini de $e = c_1a_1c_2a_2c_3a_3\ldots$, dont la configuration initiale est c_2 et la configuration finale est c_3. Le facteur $f' = c_3a_3\ldots$ est un facteur infini de e dont la configuration initiale est c_3. L'ensemble des facteurs de $\mathcal{S}$ est noté $\mathcal{F}$. L'ensemble des facteurs de $\mathcal{S}$ dont la configuration initiale est c est notée $\mathcal{F}_c$. L'ensemble des facteurs finis de $\mathcal{S}$ dont la configuration initiale est α et dont la configuration finale est β est notée $\mathcal{F}_{\alpha,\beta}$. Notons qu'un facteur fini peut être réduit à une unique configuration.

Définition 2.16 (Journal) *Un* journal *de $f \in \mathcal{F}$ est la projection de f sur les configurations qui la constituent.*

Par exemple, pour $f = c_1a_1c_2a_2c_3a_3\ldots$, le journal correspondant est $j = c_1c_2c_3\ldots$ et la configuration initiale de j est c_1. L'ensemble des journaux de $\mathcal{S}$ est noté $\mathcal{J}$. L'ensemble des journaux de $\mathcal{S}$ dont la configuration initiale est c est notée $\mathcal{J}_c$. L'ensemble des journaux de $\mathcal{S}$ dont la configuration initiale fait partie d'un ensemble $C_1 \subset \mathcal{C}$ est notée $\mathcal{J}_{C_1}$.

Définition 2.17 (Journal Restreint) *Un* journal restreint $j_{\{c_1,\ldots,c_n\}}$ *de $f \in \mathcal{F}$ est la projection de f sur les configurations $c_1, \ldots, c_n$ qui la constituent.*

Par exemple, pour $f = c_1a_1c_2a_2c_3a_3\ldots$, le journal restreint sur les configurations c_1, c_2 correspondant est $j_{\{c_1,c_2\}} = c_1c_2\ldots$ et la configuration initiale de j est c_1. L'ensemble des journaux restreints de $\mathcal{S}$ à l'ensemble $\{c_1, \ldots, c_n\}$ est noté $\mathcal{J}^{\{c_1,\ldots,c_n\}}$.

Définition 2.18 (Trace) *Une* trace *de $f \in \mathcal{F}$ est la projection de f sur les actions qui la constituent.*

Par exemple, pour $f = c_1a_1c_2a_2c_3a_3\ldots$, la trace correspondante est $t = a_1a_2a_3\ldots$ et l'action initiale de t est a_1. L'ensemble des traces de $\mathcal{S}$ est noté $\mathcal{T}$. L'ensemble des traces de $\mathcal{S}$ dont l'action initiale est a est notée $\mathcal{T}_a$.

Définition 2.19 (Trace Restreinte) *Une* trace restreinte $t_{\{a_1,\ldots,a_n\}}$ *de $f \in \mathcal{F}$ est la projection de f sur les actions $a_1, \ldots, a_n$ qui la constituent.*

Par exemple, pour $f = c_1a_1c_2a_2c_3a_3\ldots$, la trace restreinte sur les actions a_1, a_3 correspondante est $t_{\{a_1,a_3\}} = a_1a_3\ldots$ et l'action initiale de $t_{\{a_1,a_3\}}$ est a_1. L'ensemble des traces restreintes de $\mathcal{S}$ à l'ensemble $\{a_1, \ldots, a_n\}$ est noté $\mathcal{T}^{\{a_1,\ldots,a_n\}}$.

Notations. Une configuration $\beta \in \mathcal{C}$ est *accessible* depuis une configuration $\alpha \in \mathcal{C}$ si et seulement si il existe un facteur $f \in \mathcal{F}_{\alpha,\beta}$. Cette relation est notée $\alpha \rightsquigarrow \beta$.

Une action composée $A = \langle a_1 \ldots a_k\rangle \rightarrow \langle a_{k+1} \ldots a_n\rangle$ est *activable dans une configuration* $c \in \mathcal{C}$ si et seulement si il existe un facteur $f \in \mathcal{F}_c$ tel que $a_1 \ldots a_n$ est une trace restreinte à $\{a_1, \ldots, a_n\}$ de f. Dans la suite, *Activable*(A, c) renvoie *vrai* si l'action A est activable dans la configuration c.

2.4.3 Equité

Dans la pratique, la condition de maximalité des exécutions d'un système réparti $\mathcal{S}$ est insuffisante pour beaucoup d'applications. Les exécutions infinies d'un système réparti peuvent satisfaire plusieurs propriétés supplémentaires, en rapport avec la *nature* des actions rencontrées au cours d'une exécution infinie :

1. Une exécution infinie est *faiblement équitable* si il n'existe aucun suffixe non vide tel qu'une action composée A n'est jamais exécutée alors que dans toute configuration du suffixe, A est activable. Ceci revient à dire que lorsqu'une action composée d'un processeur reste continuellement activable, alors l'instruction correspondant à *cette* action composée est exécutée au bout d'un temps fini. L'ensemble des exécutions faiblement équitables de $\mathcal{S}$ est noté $FEq(\mathcal{E})$.
2. Une exécution infinie est *équitable* si il n'existe aucun suffixe non vide tel qu'une action composée A n'est jamais exécutée alors que dans une infinité de configurations du suffixe, A est activable. Ceci revient à dire que lorsqu'une action composée d'un processeur est infiniment souvent activable, alors l'instruction correspondant à *cette* action composée est exécutée au bout d'un temps fini. L'ensemble des exécutions équitables de $\mathcal{S}$ est noté $Eq(\mathcal{E})$.

Il est clair que pour tout système $\mathcal{S}$ donné, on a $Eq(\mathcal{E}) \subseteq FEq(\mathcal{E}) \subseteq \mathcal{E}$. On peut montrer que ces ensembles sont différents. Prenons l'exemple du système S décrit figure 2.13, et montrons que les inclusions sont strictes.

Lemme 2.1 $\mathrm{FEq}(\mathcal{E}) \neq \mathcal{E}$

Preuve: L'exécution

$$\Big(\ [1,1,1](\mathit{Lit}(R_1,1),-,\mathit{Lit}(R_1,1))[2,1,1](\mathit{Ecrit}(R_1,1),-,\mathit{Ecrit}(R_1,1))\ \Big)^{\omega}$$

est une exécution de S, puisque l'action gardée de P_1 est exécutée infiniment souvent.

Cependant, elle n'est pas faiblement équitable, car dans chacune des configurations $[1,1,1]$ et $[2,1,1]$, l'action gardée de P_2 est activable (car spontanée). Donc il existe un suffixe non vide où aucune instruction de P_2 n'est exécutée (P_2 n'exécute jamais $\mathit{Ecrit}(R_1,2);\mathit{Ecrit}(R_1,1)$) alors que dans toute configuration du suffixe (les configurations $[1,1,1]$ et $[2,1,1]$), l'action gardée de P_2 est activable. □

Lemme 2.2 $\mathrm{Eq}(\mathcal{E}) \neq \mathrm{FEq}(\mathcal{E})$

Preuve: L'exécution

$$\Big(\ [1,1,1](-,\mathit{Ecrit}(R_1,2),\mathit{Ecrit}(R_1,2))[1,2,2](-,\mathit{Ecrit}(R_1,1),\mathit{Ecrit}(R_1,1))\ \Big)^{\omega}$$

est une exécution faiblement équitable puisque l'action composée de P_2, est spontanée et exécutée infiniment souvent.

Cependant, elle n'est pas équitable puisque l'action composée de P_1, même si elle n'est pas constamment activable, est activable infiniment souvent au cours de l'exécution (à chaque occurrence de la configuration $[1, 1, 1]$) mais n'est jamais exécutée. □

Il en résulte que si une propriété est vraie pour toute exécution d'un algorithme, elle reste vraie dans toute exécution faiblement équitable du même algorithme, et *a fortiori* dans toute exécution équitable. Par contre, si seules les exécutions équitables d'un algorithme garantissent qu'une propriété est vérifiée, alors il n'est pas du tout garanti que cette propriété reste vraie pour toute exécution faiblement équitable de l'algorithme.

2.4.4 Démons

Un *démon* est une propriété des exécutions des systèmes répartis. Cette propriété concerne l'*ordre* des actions apparaissant dans les exécutions des systèmes. Dans la suite, on considère que l'ensemble des règles gardées d'un système réparti est donné sous la forme :

$$\begin{aligned} \langle l_1^1 \dots l_1^{i_1} \rangle &\rightarrow \langle e_1^1 \dots e_1^{i_1} \rangle \\ &\vdots \\ \langle l_n^1 \dots l_n^{i_n} \rangle &\rightarrow \langle e_n^1 \dots e_n^{i_n} \rangle \end{aligned}$$

Les contraintes d'*atomicité* concernent l'*ordre* des actions apparaissant dans l'ensemble des traces :

1. Un démon à *lecture-écriture* (noté $\mathcal{D}_{LE}$) n'impose pas d'ordre particulier sur les actions.
2. Un démon *totalement réparti* (noté $\mathcal{D}_{TR}$) impose que les actions de lecture relatives à une même règle gardée soient groupées, et que les actions d'écriture relatives à une même règle gardée soient groupées. Ce démon considère en fait que les gardes sont évaluées atomiquement, et que les actions composées associées sont exécutées atomiquement. Formellement, cela signifie que toute trace d'un système sous le démon totalement réparti est une séquence $a_0, a_1 \dots$ où chaque a_k est soit de la forme $l_j^1, \dots, l_j^{i_j}$ soit de la forme $e_j^1, \dots, e_j^{i_j}$, pour $n \geq j \geq 1$.
3. Un démon est *réparti* (noté $\mathcal{D}_R$) si et seulement si il est totalement réparti et toute trace peut être organisée en une séquence de phase d'évaluation des gardes puis de phase d'exécution des actions composées associées. Formellement, cela signifie que toute trace d'un système sous le démon réparti est une séquence $a_0, a_1, \dots$ où chaque $a_{2\times k}$ est de la forme

   $$l_{j_1}^1, \dots, l_{j_1}^{i_{j_1}}, \dots, l_{j_p}^1, \dots, l_{j_p}^{i_{j_p}}$$

   (avec $n \geq p \geq 1$), et où chaque $a_{2\times k+1}$ est de la forme

   $$e_{j_1}^1, \dots, e_{j_1}^{i_{j_1}}, \dots, e_{j_p}^1, \dots, e_{j_p}^{i_{j_p}}$$

   (avec le même p).

4. Un démon est *synchrone* (noté $\mathcal{D}_S$) si et seulement si il est réparti et toute séquence de phases implique tous les processeurs du système. Formellement, cela signifie que toute trace d'un système sous le démon synchrone est une séquence $a_0, a_1, \ldots$ où chaque $a_{2\times k}$ est de la forme
$$l^1_{j_1}, \ldots, l^{i_{j_1}}_{j_1}, \ldots, l^1_{j_n}, \ldots, l^{i_{j_n}}_{j_n}$$
et où chaque $a_{2\times k+1}$ est de la forme
$$e^1_{j_1}, \ldots, e^{i_{j_1}}_{j_1}, \ldots, e^1_{j_n}, \ldots, e^{i_{j_n}}_{j_n}$$

5. Un démon est *centralisé* (noté $\mathcal{D}_C$) si et seulement si il est réparti et toute séquence de phase implique un unique processeur du système. Formellement, cela signifie que toute trace d'un système sous le démon centralisé est une séquence $a_0, a_1, \ldots$ où chaque $a_{2\times k}$ est de la forme $l^1_j, \ldots, l^{i_j}_j$, et où chaque $a_{2\times k+1}$ est de la forme $e^1_j, \ldots, e^{i_j}_j$.

Définition 2.20 (Démon) *L'ensemble des exécutions du système $\mathcal{S}$ sous le démon $\mathcal{D}$ est un sous-ensemble $\mathcal{E}_\mathcal{D}$ de l'ensemble $\mathcal{E}$ des exécutions de $\mathcal{S}$, tel que chaque trace d'une exécution $e \in \mathcal{E}_\mathcal{D}$ satisfait $\mathcal{D}$.*

Un démon A est *plus puissant* qu'un démon B si et seulement si $\mathcal{E}_B \subset \mathcal{E}_A \wedge \mathcal{E}_B \neq \mathcal{E}_A$.

Il est clair que $\mathcal{E}_{\mathcal{D}_S} \subseteq \mathcal{E}_{\mathcal{D}_R}$, $\mathcal{E}_{\mathcal{D}_C} \subseteq \mathcal{E}_{\mathcal{D}_R}$ et $\mathcal{E}_{\mathcal{D}_R} \subseteq \mathcal{E}_{\mathcal{D}_{TR}} \subseteq \mathcal{E}_{\mathcal{D}_{LE}}$. On peut montrer que tous les démons sont différents les uns des autres. Considérons un système $\mathcal{S}$ constitué de trois processeurs P_1, P_2 et P_3 capables de communiquer au moyen de trois registres R_1, R_2 et R_3. Le code de chaque processeur est donné par les règles suivantes, où l^r_p désigne une action de lecture du processeur P_p sur le registre R_r et où e^r_p désigne une action d'écriture du processeur P_p sur le registre R_r :

$$\begin{array}{lcl} P_1 & :: & \langle l^1_1 l^2_1 \rangle \rightarrow \langle e^1_1 \rangle \\ & & \langle l^2_1 l^3_1 \rangle \rightarrow \langle e^2_1 \rangle \\ P_2 & :: & \langle l^1_2 \rangle \rightarrow \langle e^1_2 e^2_2 \rangle \\ P_3 & :: & \langle l^1_3 \rangle \rightarrow \langle e^3_3 \rangle \end{array}$$

Montrons en exhibant des traces particulières du système $\mathcal{S}$ que les démons sont différents.

Lemme 2.3 $\mathcal{E}_{\mathcal{D}_S} \neq \mathcal{E}_{\mathcal{D}_R}$

Preuve: La trace $l^1_1 l^2_1 l^1_2 e^1_1 e^1_2 e^2_2 l^1_3 e^1_3 \ldots$ est élément de $\mathcal{D}_R$ (le démon réparti) mais pas de $\mathcal{D}_S$ (le démon synchrone) car la première séquence de phases n'inclut pas le processeur P_3. □

Lemme 2.4 $\mathcal{E}_{\mathcal{D}_C} \neq \mathcal{E}_{\mathcal{D}_R}$

Preuve: La trace $l^1_1 l^2_1 l^1_2 e^1_1 e^1_2 e^2_2 l^1_3 e^1_3 \ldots$ est élément de $\mathcal{D}_R$ (le démon réparti) mais pas de $\mathcal{D}_C$ (le démon central) car la première séquence de phases inclut deux processeurs (P_1 et P_2). □

Lemme 2.5 $\mathcal{E}_{\mathcal{D}_R} \neq \mathcal{E}_{\mathcal{D}_{TR}}$

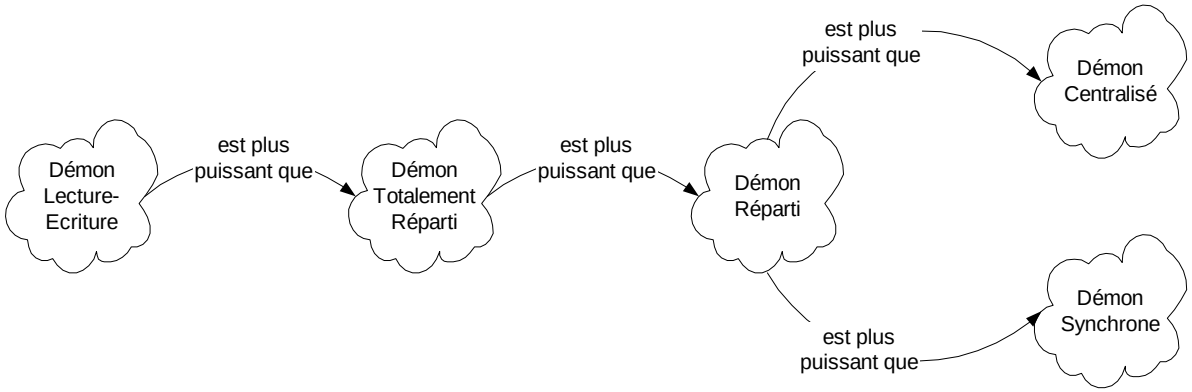

FIG. 2.14 – Hiérarchie des Démons

Preuve: La trace $l_1^1 l_1^2 l_2^1 e_1^1 l_3^1 e_3^1 e_2^1 e_2^2 \ldots$ est élément de $\mathcal{D}_{TR}$ (le démon totalement réparti) mais pas de $\mathcal{D}_R$ (le démon réparti) car la phase d'exécution des actions composées associées aux gardes $\langle l_1^1 l_1^2 \rangle$ et $\langle l_2^1 \rangle$ n'est pas atomique mais interrompue par l'exécution de la garde $\langle l_3^1 \rangle$ puis de l'action $\langle e_3^1 \rangle$. □

Lemme 2.6 $\mathcal{E}_{\mathcal{D}_{TR}} \neq \mathcal{E}_{\mathcal{D}_{LE}}$

Preuve: La trace $l_1^1 l_3^1 e_3^1 l_1^2 l_2^1 e_1^1 e_2^1 e_2^2 \ldots$ est élément de $\mathcal{D}_{LE}$ (le démon lecture-écriture) mais pas de $\mathcal{D}_{TR}$ (le démon totalement réparti) car la phase d'exécution des actions composées associées à la garde $\langle l_1^1 l_1^2 \rangle$ n'est pas atomique mais interrompue par l'exécution de la garde $\langle l_3^1 \rangle$ puis de l'action $\langle e_3^1 \rangle$. □

La hiérarchie des démons que nous venons de présenter se trouve résumée sur la figure 2.14. Il en résulte que si une propriété est vraie pour toute exécution d'un algorithme sous le démon lecture-écriture, elle reste vraie dans toute exécution du même algorithme sous le démon réparti, et *a fortiori* dans toute exécution sous le démon central. Par contre, si seules les exécutions d'un algorithme sous le démon synchrone garantissent qu'une propriété est vérifiée, alors il n'est pas du tout garanti que cette propriété reste vraie pour toute exécution de l'algorithme sous le démon totalement réparti.

2.4.5 Spécification

Résoudre une tâche implique d'être en mesure de la décrire avec précision, c'est à dire de la *spécifier*. Une *spécification* est une famille d'ensembles d'exécutions possédant une propriété commune. Chaque ensemble d'exécutions se rapporte à un système particulier résolvant la tâche spécifiée. Dans cette thèse, les tâches que nous souhaitons résoudre peuvent être spécifiées au moyen d'*exécutions abstraites*, qui sont des classes d'exécutions indépendantes des systèmes qui implantent les tâches. Une exécution abstraite est une séquence, finie ou infinie, de *configurations abstraites*. Une configuration abstraite est un vecteur des états d'un ensemble de *variables abstraites*. Une variable abstraite caractérise la tâche à résoudre.

Exemple Considérons la tâche de l'*exclusion mutuelle* dans un système synchrone, où tous les processeurs fonctionnent à la même vitesse. Pour définir la tâche consistant à maintenir une exclusion mutuelle dans un système réparti synchrone, on peut supposer qu'une variable abstraite nommée *Priv* est associée à chaque processeur. Cette variable peut avoir pour valeur *vrai* ou *faux* et est caractéristique du fait que le processeur est en train d'exécuter sa section critique. Dès lors, la tâche consistant à maintenir une exclusion mutuelle est satisfaite à partir du moment où, à chaque instant, au plus une variable *Priv* est à *vrai*. Définir la tâche consistant à garantir une exclusion mutuelle *équitable* revient à considérer que non seulement à chaque instant au plus une variable *Priv* est à vrai, mais que de plus au cours du temps, chaque variable *Priv* est à *vrai* infiniment souvent. Les expressions "à chaque instant" et "au cours du temps" renvoient aux concepts de configurations abstraites (les valeurs à un instant donné des variables abstraites) et d'exécutions abstraites (les séquences de configurations abstraites).

Définition 2.21 (Spécification) *La* spécification *d'une tâche est un ensemble (possiblement infini) d'exécutions abstraites.*

Munis de spécification d'une tâche T, nous désirons savoir si un système réparti $\mathcal{S}$ donné résout T (on dit aussi que $\mathcal{S}$ satisfait la spécification $\mathcal{P}$ de la tâche T). Ainsi, nous définissons la fonction $Abs_{\mathcal{P}}$ sur les configurations d'un système réparti $\mathcal{S}$, qui renvoie la configuration abstraite associée à la configuration de $\mathcal{S}$.

Exemple Considérons un système S de n processeurs sans variables locales, où le processeur P_i exécute un code composé de a_i actions atomiques (pour tout i, on suppose $a_i \geq 3$). L'état du processeur P_i est décrit par son pointeur de programme (qui varie de 1 à a_i). Sa fonction de transition induit une boucle sans fin des actions 1 à a_i. Chaque processeur P_i possède une action atomique particulière (l'action 2) qui doit être exécutée en section critique (c'est à dire qu'une seule de ces actions doit être exécutée à la fois dans le système tout entier). Il est alors possible de définir la fonction $Abs_{\mathcal{P}}$ sur les configurations de S_1 pour la tâche d'exclusion mutuelle. Cette fonction associe à chaque processeur P_i une variable $Priv_i$ qui vaut *vrai* si et seulement si l'état de P_i vaut 2. Ainsi, si à chaque instant, au plus un processeur du système a la possibilité d'exécuter son action 2, alors à chaque instant au plus une variable $Priv_i$ est à *vrai*, la tâche consistant à maintenir une exclusion mutuelle est effectuée.

Par extension, la fonction $Abs_{\mathcal{P}}$ est étendue aux journaux de $\mathcal{S}$ en appliquant $Abs_{\mathcal{P}}$ à chaque configuration du journal considéré, pour obtenir une exécution abstraite. La spécification d'une tâche pour un système est alors une condition sur les exécutions d'un système réparti, qui porte sur les *configurations* du système.

Exemple Considérons le même système S que dans l'exemple précédent. Si au cours de toutes les exécutions possibles de S_1, au plus un processeur est dans l'état 2 et que tous les processeurs sont dans l'état 2 infiniment souvent, alors par le biais de la même fonction

$Abs_{\mathcal{P}}$, les exécutions abstraites associées aux journaux de S garantissent qu'une exclusion mutuelle équitable est maintenue par le système S.

Définition 2.22 (Vérifier une Spécification) *L'ensemble des exécutions du système $\mathcal{S}$ vérifiant la spécification $\mathcal{P}$ est un sous-ensemble $\mathcal{E}_{\mathcal{P}}$ de l'ensemble $\mathcal{E}$ des exécutions de $\mathcal{S}$ tel que pour chaque journal j d'une exécution $e \in \mathcal{E}_{\mathcal{P}}$, $\mathrm{Abs}_{\mathcal{P}}(j) \in \mathcal{P}$.*

Un système *vérifie la spécification* $\mathcal{P}$ si et seulement si $\mathcal{E} \subset \mathcal{E}_{\mathcal{P}}$. On dit alors que toute exécution $e \in \mathcal{E}$ est *légitime*.

Le plus souvent, une spécification est décrite au moyens de prédicats sur les configurations abstraites. Un tel prédicat est *vrai* si la propriété est vérifiée, et *faux* sinon. Pour obtenir, à partir des prédicats sur les configurations abstraites, des prédicats sur les exécutions abstraites, il suffit de donner des conditions sur les valeurs des prédicats dans les configurations abstraites rencontrées au cours de l'exécution abstraite. On distingue dans la littérature deux grands types de prédicats sur les exécutions abstraites :

1. *Sureté* : dans toute configuration abstraite de toute exécution abstraite, le prédicat P n'est jamais vérifié. De manière informelle, on cherche à s'assurer qu'une propriété non voulue n'arrive jamais.
2. *Vivacité* : dans toute exécution, le prédicat P est vrai dans au moins une configuration. De manière informelle, on cherche à s'assurer qu'une propriété voulue finit par arriver. Une variante de la spécification de vivacité est la spécification d'équité (ou de vivacité forte) : dans toute exécution, le prédicat P est vrai une infinité de fois. Notons que la *spécification d'équité* est différente de l'*hypothèse d'équité* des exécutions du système (définie dans la section 2.4.3).

Une spécification $\mathcal{P}$ est *statique* si toute exécution abstraite de $\mathcal{P}$ pour un système $\mathcal{S}$ possède un suffixe non vide de configurations abstraites identiques. Soit $\mathcal{C}_{\mathcal{P}}$ l'ensemble des configurations abstraites de chacun de ces suffixes : $\mathcal{C}_{\mathcal{P}}$ est appelé ensemble des *configurations légitimes* de $\mathcal{P}$. Le complémentaire dans $\mathcal{C}$ de $\mathcal{C}_{\mathcal{P}}$ est l'ensemble des configurations *illégitimes*.

Une spécification $\mathcal{P}$ est *dynamique* si au moins une exécution abstraite de $\mathcal{P}$ pour un système $\mathcal{S}$ est infinie et ne possède aucun suffixe non vide de configurations abstraites identiques.

2.5 Auto-stabilisation

2.5.1 Définition

Dans les systèmes répartis habituels (c'est à dire non-stabilisants), il est possible de restreindre l'ensemble des exécutions du système en autorisant celles-ci à ne commencer que dans un ensemble restreint des configurations appelées *configurations initiales*. Les systèmes répartis auto-stabilisants ne tolèrent pas ce genre de facilités puisque toutes les configurations possibles du système sont des configurations initiales admissibles.

Définition 2.23 (Auto-stabilisation) *Le système $\mathcal{S}$ est* auto-stabilisant *pour la spécification $\mathcal{P}$ si et seulement si il existe un sous-ensemble $\mathcal{C}_\mathcal{L}$ de l'ensemble $\mathcal{C}$ des configurations de $\mathcal{S}$ tel que : (i) toute exécution de $\mathcal{S}$ dont la configuration initiale appartient à $\mathcal{C}_\mathcal{L}$ vérifie $\mathcal{P}$ et (ii) toute exécution de $\mathcal{S}$ contient au moins une configuration appartenant à $\mathcal{C}_\mathcal{L}$.*

Dans la littérature, $\mathcal{C}_\mathcal{L}$ est souvent référencé par le terme de *configuration légitime*, la condition (i) par le terme de *correction*, et la condition (ii) par le terme de *convergence*.

Le concept de *pseudo-stabilisation* a été introduit dans [BGM93] et présente un affaiblissement de la définition de l'auto-stabilisation. C'est à dire que tout système auto-stabilisant est également pseudo-stabilisant, mais que la réciproque n'est pas vraie. Dans la pratique, les systèmes pseudo stabilisants sont suffisants pour la plupart des tâches à résoudre.

Définition 2.24 (Pseudo-stabilisation) *Le système $\mathcal{S}$ est* pseudo-stabilisant *pour la spécification $\mathcal{P}$ si et seulement si toute execution e de l'ensemble $\mathcal{E}$ des exécutions de $\mathcal{S}$ possède un suffixe non vide qui vérifie $\mathcal{P}$.*

Le concept de stabilisation instantanée formalise le fait que si toutes les configuration du système sont légitimes, alors le temps de stabilisation est nul. En pratique, si un système statique est instantanément stabilisant, alors il peut être trivialement résolu ; par contre, les systèmes dynamiques instantanément stabilisants sont intéressants au titre qu'ils supportent, en plus des défaillances transitoires sur les processeurs et les liens, des changement de topologie inopinés tout au long de l'exécution.

Définition 2.25 (Stabilisation Instantanée) *Le système $\mathcal{S}$ est* instantanément stabilisant *pour la spécification $\mathcal{P}$ si et seulement si toute exécution e de l'ensemble $\mathcal{E}$ des exécutions de $\mathcal{S}$ vérifie $\mathcal{P}$.*

2.5.2 Prouver l'Auto-stabilisation

Prouver la correction des systèmes auto-stabilisants ne diffère pas des preuves de correction que l'on peut trouver pour les algorithmes répartis classiques : on part d'un ensemble de configurations (les configurations initiales dans le cas des systèmes non-stabilisants ; les configuration légitimes dans le cas des systèmes auto-stabilisants) et on prouve que toute exécution abstraite effectuée à partir de ces configurations satisfait la spécification de la tâche impartie au système. Les technique habituelles d'invariant peuvent être utilisées.

Prouver la convergence est moins classique puisqu'il faut montrer que de chacune des configurations *possibles* du système (c'est à dire de chaque produit possible des états des composants de base du système), toute exécution mène à une configuration légitime. Plusieurs méthodes principales sont proposées dans la littérature : la *fonction décroissante*, les *attracteurs*, la *réécriture* ([BBF99]). Dans le reste de la thèse, nous utilisons principalement les deux premières.

Fonction Décroissante

Afin de prouver la convergence d'un système réparti vers un ensemble de configurations, nous définissons une fonction à valeurs entières positives sur les configurations du système. Cette fonction doit avoir la particularité d'atteindre une valeur minimale dans chacune des configurations légitimes, et de décroître strictement à chaque exécution d'une action du système à partir d'une configuration non légitime. Puisque l'ensemble des entiers positifs est bien ordonné, une telle fonction garantit qu'à partir de toute configuration du système, et pour toute exécution possible à partir de cette configuration, le système finit par atteindre une configuration légitime (il n'existe pas de suite strictement décroissante non bornée inférieurement dans $\mathbb{N}$).

Attracteurs

Pour les systèmes répartis complexes où une fonction décroissante intuitive est difficile à trouver, il est possible d'utiliser une variante sous la forme d'*attracteurs* de configurations. Intuitivement, les attracteurs sont des ensembles de configurations tels qu'à partir de toute configuration d'un ensemble donné, toute exécution atteint fatalement une configuration de l'attracteur. En quelque sorte, les attracteurs permettent de définir des paliers de convergence : le système satisfait tout d'abord une propriété p, puis si la propriété p est vérifiée, alors la propriété p', plus restrictive que p est à son tour vérifiée, et ainsi de suite.

Considérons un système $\mathcal{S}$ dont l'ensemble des configurations est $\mathcal{C}$, et dont l'ensemble des exécutions sous le démon $\mathcal{D}$ est $\mathcal{E}$.

Définition 2.26 (Attracteur) *Soit C_a et C_b deux sous-ensembles de $\mathcal{C}$. L'ensemble C_a est un* attracteur *pour l'ensemble C_b si et seulement si pour toute configuration initiale $c_1 \in C_b$, et pour toute exécution $e \in \mathcal{E}_{c_1}$, $e = c_1 a_1 c_2 a_2 \ldots$, il existe $i \geq 1$ tel que $c_i \in C_a$. L'ensemble C_a est un* attracteur clos *pour l'ensemble C_b si et seulement si pour toute configuration initiale $c_1 \in C_b$, et pour toute exécution $e \in \mathcal{E}_{c_1}$, $e = c_1 a_1 c_2 a_2 \ldots$, il existe $i \geq 1$ tel que pour tout $j \geq i$, $c_j \in C_a$.*

Prouver l'auto-stabilisation avec des attracteurs revient alors à trouver des ensembles de configurations $C_1, \ldots, C_n$ tels que pour tout $i \in \{2, \ldots, n\}$, C_i est un attracteur pour C_{i-1}, et tels que $C_1 = \mathcal{C}$ et C_n est l'ensemble des configurations légitimes.

2.6 Complexité

2.6.1 Complexité en Espace

La complexité en espace d'un algorithme est simplement la taille en nombre de bits des variables utilisées par chaque processeur ainsi que la taille des registres (ou des messages dans le cas où les liens sont des câbles) utilisés par les processeurs pour communiquer entre eux. On peut classifier l'efficacité en mémoire d'un système réparti selon la catégorie de taille mémoire utilisée sur chaque composant :

1. *Constante* : la taille mémoire utilisée par chacun des composants de base est indépendante de la taille du système.
2. *Degré* : la taille mémoire utilisée par chacun des composants de base est seulement dépendante du degré maximal du graphe de communication du système. La technologie actuelle limite *de facto* le degré des réseaux d'ordinateurs ou de processeurs, ce qui assimile en pratique cette classe de complexité mémoire à la classe constante.
3. *Logarithmique* : la taille mémoire est fonction de la taille du système, mais compte tenu de la taille des systèmes actuellement constructibles, il est possible de dimensionner la mémoire utilisée de manière à ce qu'elle reste constante sur tout système actuel.
4. *Polynomiale* : la taille mémoire est fonction de la taille du système, donc les variables utilisées augmentent polynomialement avec la taille du système. Ceci limite l'implantation de tels système sur des réseaux constitués de quelques milliers de machines compte tenu des mémoire actuelles.
5. *Exponentielle* : la taille mémoire est fonction de la taille du système, mais les variables utilisées sont tellement importantes que l'implantation se trouve limitée de fait à des réseaux de très petite taille (machines parallèles).

2.6.2 Complexité en Temps

Du fait que les différents composants du système évoluent à des vitesses différentes, il est difficile de définir un temps global absolu permettant d'évaluer si la tâche est résolue rapidement ou pas, ou de comparer deux implantations de systèmes répartis destinés à résoudre une même tâche.

Mesurer le Temps

Habituellement, la complexité en temps des algorithmes répartis s'effectue en comptant le nombre de messages échangés au cours de l'exécution de l'algorithme. Cette complexité apparaît mal adaptée au cas particulier des algorithmes répartis auto-stabilisants, pour deux raisons principales :

1. Dans les systèmes où la communication s'effectue par passage de message, des défaillances transitoires peuvent faire en sorte qu'un nombre non borné de messages erronés soit en transit dans les canaux au départ de l'exécution, ce qui ne permet pas de donner une borne sur le nombre global de messages échangés ;
2. Certains systèmes auto-stabilisants ne fonctionnent correctement que si on suppose leur exécutions équitables ou faiblement équitables, ce qui signifie qu'un nombre arbitrairement grand de messages peuvent être échangés.

Afin de définir un temps logique pour les systèmes répartis auto-stabilisants et d'être en mesure d'établir des bornes de complexité en temps, nous utilisons dans cette thèse la notion de cycle asynchrone d'exécution. Dans un cycle asynchrone d'exécution à partir d'une configuration c, tout processeur du système réparti dont au moins une action gardée

est activable dans c effectue au moins une action gardée si celle-ci est constamment activable pendant le cycle asynchrone d'exécution.

Etant donné un système réparti, il est possible de déterminer plusieurs grandeurs temporelles ayant trait aux exécutions du système au moyen des cycles asynchrones d'exécution :

1. *Temps de stabilisation* : le temps (*i.e.* le nombre de cycles asynchrones d'exécution) maximum nécessaire pour atteindre une configuration légitime à partir d'une configuration quelconque.
2. *Temps de calcul* : le temps maximum nécessaire pour atteindre une configuration légitime d'une tâche statique à partir d'une configuration à connaissances purement locales.
3. *Temps de service* : temps maximum nécessaire entre deux occurrences d'une même configuration légitime pour une tâche dynamique. Notons que dans le calcul du temps de service, seules des configurations légitimes apparaissent dans l'exécution considérée.

Exemple pour une Tâche Statique Définissons la tâche statique du calcul de la taille d'un réseau. Chaque processeur P_i (pour i variant de 1 à n) est associé à une variable abstraite $Taille_i$ et une configuration est légitime si et seulement si toutes les variables *Taille* sont égales à n.

Le temps de stabilisation d'un système réparti satisfaisant la spécification du calcul de taille est le nombre maximum de cycles asynchrones d'exécutions nécessaire pour atteindre une configuration légitime à partir de n'importe quelle configuration du système. En particulier, la variable abstraite associée à chaque processeur peut prendre des valeurs complètement fausse dans une configuration initiale, comme $4 \times n + 3$ ou encore $7 \times n + 2$.

Le temps de calcul d'un système réparti satisfaisant la spécification du calcul de taille est le nombre maximum de cycles asynchrones d'exécutions nécessaire pour atteindre une configuration légitime à partir d'une configuration où chaque processeur P_i ne connaît rien du reste du réseau. La variable $Taille_i$ qui lui est associée vaut 1 (le processeur pense qu'il est l'unique processeur du réseau).

Exemple pour une Tâche Dynamique Le temps de stabilisation d'un système réparti satisfaisant la spécification de l'exclusion mutuelle équitable (voir page 40) est le nombre maximum de cycles asynchrones d'exécutions nécessaire pour atteindre une configuration légitime à partir de n'importe quelle configuration du système, c'est à dire une configuration où au plus une unique variable abstraite $Priv_i$ vaut *vrai*.

Le temps de service d'un système réparti satisfaisant la spécification de l'exclusion mutuelle équitable est le nombre maximum de cycles asynchrones d'exécutions nécessaire pour revenir deux fois dans la même configuration abstraite légitime.

Mesurer le Surcoût

Le temps de stabilisation est la principale grandeur considérée dans la littérature. Dans cette thèse, nous considérons également le temps de calcul et le temps de service. Ces deux

mesures ont trait au *surcoût* occasionné par l'utilisation d'un algorithme auto-stabilisant par rapport à un système non-stabilisant.

Etant donné une tâche statique, un algorithme non-stabilisant A donne un résultat en l'absence de défaillances transitoires au bout d'un temps $calcul(A)$. De la même manière, un algorithme auto-stabilisant A' donne un résultat, toujours en l'absence de défaillances, au bout d'un temps $calcul(A')$. Maintenant, si des défaillances transitoires se produisent au cours de l'exécution du système, alors l'algorithme A (qui est non-stabilisant) peut très bien ne *jamais* fournir de résultat, tandis que l'algorithme A' (qui est auto-stabilisant) fournit toujours un résultat correct au bout d'un temps $stabilisation(A')$.

Le cas des tâches dynamiques est similaire. Etant donné une tâche dynamique, un algorithme non-stabilisant A offre un service en l'absence de défaillances transitoires au bout d'un temps $service(A)$. De la même manière, un algorithme auto-stabilisant A' offre un service, toujours en l'absence de défaillances, au bout d'un temps $service(A')$. Maintenant, si des défaillances transitoires se produisent au cours de l'exécution du système, alors l'algorithme A (qui est non-stabilisant) peut très bien ne *jamais* offrir de service, tandis que l'algorithme A' (qui est auto-stabilisant) offre toujours un service correct au bout d'un temps $stabilisation(A')+service(A')$.

Le tableau ci-dessous récapitule les possibilités offertes par les algorithmes auto-stabilisants par rapport à leurs homologues classiques.

Spécification	**Défaillances**	A **non-stabilisant**	A' **stabilisant**
statique	non	*calcul* (A)	*calcul* (A')
	oui	**pas de résultat**	$stabilisation(A')$
dynamique	non	*service* (A)	*service* (A')
	oui	**pas de service**	*stabilisation* (A')+*service* (A')

Dans la suite, nous considérons que la version auto-stabilisante d'un algorithme réparti classique est *sans surcoût* si son temps de calcul ou de service (suivant le type de tâche considéré) est identique à celui de la version non-stabilisante. Le temps de stabilisation reste un bon indicateur de la performance du système en présence de défaillances transitoires.

2.7 Résumé

Nous modélisons un système réparti par un automate résultant d'un produit synchronisé d'automates de base représentant les processeurs et les liens de communication. Différentes projections des exécutions sont proposées : traces, journaux, facteurs. Par la suite, plusieurs propriétés sur les exécutions des systèmes répartis sont explicitées. Par exemple, les propriétés portant sur l'ordre des actions apparaissant dans les exécutions permettent de classifier les différents types de démons trouvés dans la littérature, tandis que les propriétés portant sur l'apparition d'actions dans une exécution permettent de classifier les différentes hypothèses d'équité.

Un système réparti est auto-stabilisant si indépendemment de sa configuration initiale, chacune de ses exécutions converge vers un comportement correct au bout d'un temps fini. Quand ce temps est nul, le système est instantanément stabilisant.

L'efficacité d'un système auto-stabilisant est défini suivant deux critères : son temps de stabilisation en présence de défaillances transitoires et son surcoût en l'absence de défaillances.

Chapitre 3

Détecter les Défaillances

Dans un système où peuvent survenir des défaillances transitoires, il est impossible à un processeur de "savoir" si le système se trouve dans un état cohérent. En effet, supposons que chaque processeur dispose d'une variable booléenne qui est *vraie* si et seulement si le système se trouve dans un état correct. La valeur de cette variable peut être elle-même corrompue et donc ne pas refléter la situation du système. C'est la raison pour laquelle les processeurs des systèmes auto-stabilisants doivent continuer l'exécution de l'algorithme indéfiniment et ne savent jamais que le système est stabilisé. Dans ce chapitre, nous proposons un outil pour signaler des incohérences dans un système, le *détecteur de défaillances transitoires*. Chaque processeur du système dispose alors d'un *oracle* qu'il peut appeler à tout instant et qui est en mesure de détecter une faute quand le système se trouve dans un état incohérent.

Notre approche s'attache à l'implantation des détecteurs de défaillances et non à l'action à entreprendre en cas de détection d'erreur. Mentionnons cependant deux actions possibles, la réinitialisation (voir [APSVD94]) et la réparation (voir [DH97, GGHP96, AD97]).

3.1 Références

Le terme de détecteur de défaillances a été introduit dans un contexte différent dans [CT92], où un détecteur abstrait est utilisé pour résoudre le problème du consensus dans un système asynchrone. Dans le contexte des systèmes auto-stabilisants, la vérification de la consistance du système a été utilisée dans [KP93] où un instantané du système est effectué répétitivement. Des détecteurs de défaillances, appelés des observateurs, bénéficiant d'une initialisation et non sujets aux défaillances sont utilisés dans [LS92]. La vérification *locale* de consistance pour un ensemble restreint d'*algorithmes* a été suggérée dans [APSV91, AKY90]. Une technique de vérification locale pour tout algorithme, qu'il soit interactif ou non, a été présentée dans [AD97]. Cette technique est suffisamment générale pour vérifier la consistance de n'importe quel *algorithme*, et utilise des pyramides d'instantanés (ce qui nécessite une mémoire proportionelle à la taille du système sur chaque processeur).

Dans la section 3.3, nous présentons une hiérarchie de détecteurs de défaillances à la fois pour des *tâches* et pour des *algorithmes*, qui est basée sur la quantité d'information

utilisée par le détecteur de défaillances. Les détecteurs que nous présentons détectent des défaillances transitoires qui ne corrompent que l'état du système mais pas le code de chacun des processeurs. Nous distinguons une *tâche* qui est le problème à résoudre, de l'*algorithme* qui est l'implantation qui résout le problème. Les détecteurs de défaillances pour des tâches sont génériques au sens où il est possible de changer l'implantation de la tâche sans qu'il soit nécessaire de modifier le détecteur de défaillances.

De plus, nous sommes en mesure de classifier les détecteurs de défaillances en terme de *localité en distance* (dans la section 3.4) et de *localité en histoire* (dans la section 3.5) des tâches. La localité en distance est liée au diamètre de la portion de la configuration du système qu'un détecteur doit utiliser afin d'être en mesure de détecter une erreur. La localité en histoire est liée au nombre de portions successives de configurations du système qu'un détecteur doit utiliser afin d'être en mesure de détecter une défaillance transitoire. La localité en espace et la localité en temps donnent au concepteur d'algorithme des indications concernant les techniques et ressources à utiliser pour implanter la tâche.

Dans la section 3.6, nous étudions les détecteurs de défaillances pour des implantations spécifiques — c'est à dire des algorithmes spécifiques. Évidemment, tout détecteur pour une tâche peut être utilisé comme détecteur pour un algorithme implantant la tâche sans même considérer les modalités d'implantation de l'algorithme. Cependant, nous montrons que dans de nombreux cas, la quantité de ressources requise est considérablement réduite quand le détecteur de défaillances pour une tâche est remplacé par un détecteur de défaillances pour un algorithme.

3.2 Hypothèses Spécifiques au Chapitre

Dans ce chapitre, nous supposons que le graphe de communication d'un système réparti est de topologie quelconque mais non orienté (ainsi, chaque processeur du système est en mesure de prendre connaissance de l'état de chacun de ses voisins).

Les exécutions considérées sont faiblement équitables (voir section 2.4.3) et s'exécutent sous le démon lecture-écriture (voir section 2.4.4).

3.3 Détecteurs de Défaillances Transitoires

Le but d'un *détecteur de défaillances* est de vérifier qu'une configuration particulière d'un système réparti vérifie sa spécification. Plus précisément, un détecteur de défaillances est assimilé à une variable booléenne qui prend la valeur *vrai* si et seulement si la spécification est vérifiée.

Histoires et Vues Considérons un système réparti $\mathcal{S} = (P, L)$ et soit $\mathcal{G}$ son graphe de communication. La tâche résolue par $\mathcal{S}$ est spécifiée au moyen de variables abstraites associées à chaque processeur P_i de $\mathcal{S}$. Nous définissons une *vue* à partir d'un processeur P_i de $\mathcal{G}$ comme l'aggrégation *(i)* du sous-graphe de $\mathcal{G}$ centré en i et de rayon d, et *(ii)* des variables abstraites

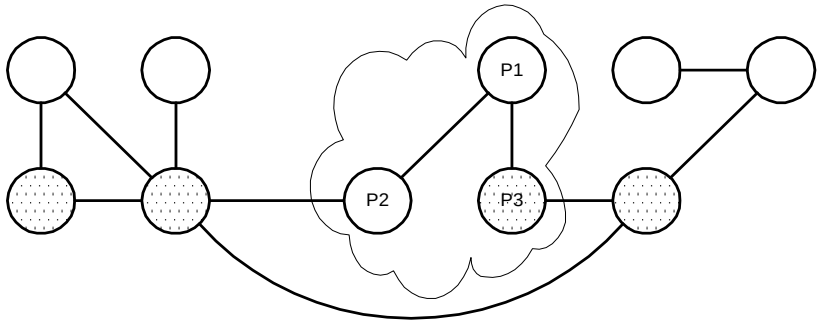

FIG. 3.1 – La vue du processeur P_1 comprend les processeurs P_1, P_2 et P_3.

associées à chacun des nœuds de ce sous-graphe. Une telle vue peut être considérée comme la projection sur la boule de rayon d centrée en i de la configuration abstraite du système $\mathcal{S}$ à un instant précis.

Définition 3.1 (Vue) *La* vue $\mathcal{V}_i^d$ *à distance d depuis un processeur P_i contient le sous-graphe de $\mathcal{G}$ contenant les nœuds de $Boule(i,d)$ ainsi que l'ensemble des variables abstraites associées à chaque nœud de ce sous-graphe pour la tâche considérée.*

Exemple Considérons un système réparti résolvant une tâche spécifiée au moyen d'une variable abstraite booléenne par processeur. La figure 3.1 présente une configuration de ce système (ou un nœud grisé représente un processeur dont la variable abstraite associée est à *faux*, et un nœud blanc un processeur dont la variable abstraite associée est à *vrai*) ainsi que la vue à distance 1 du processeur P_1. Celle-ci comprend *(i)* des informations de topologie : P_1 est connecté à deux autres processeurs P_2 et P_3, P_2 et P_3 ont un arête connectée autre que celle qui les relie à P_1, et *(ii)* des informations sur les variables abstraites associées à ces processeurs : les variables abstraites associées à P_1 et à P_2 sont à *vrai* ; la variable abstraite associée à P_3 est à *faux*.

L'*histoire* du processeur P_i est un tableau $\mathcal{V}_i^d[1..s]$ (pour des entiers d et s donnés) de s *vues* consécutives $\mathcal{V}_i^d[1]$, $\mathcal{V}_i^d[2]$, ..., $\mathcal{V}_i^d[s]$. L'élément du tableau $\mathcal{V}_i^d[j]$, pour un indice j donné, est la vue à distance d à partir du processeur P_i. L'indice j représente l'instant associé à la vue : la vue $\mathcal{V}_i^d[1]$ est associée à l'instant présent, tandis que la vue $\mathcal{V}_i^d[s]$ est associée à l'instant passé il y a $s-1$ instants.

Définition 3.2 (Histoire) *L'*histoire $\mathcal{V}_i^d[1..s]$ *à distance d depuis un processeur P_i est une séquence de s vues consécutives à distance d depuis P_i.*

Détecteurs de Défaillances Les détecteurs de défaillances que nous considérons dans ce chapitre sont répartis au sens qu'ils peuvent être questionnés par chacun des processeurs du système et donner une réponse différente à chacun d'entre eux.

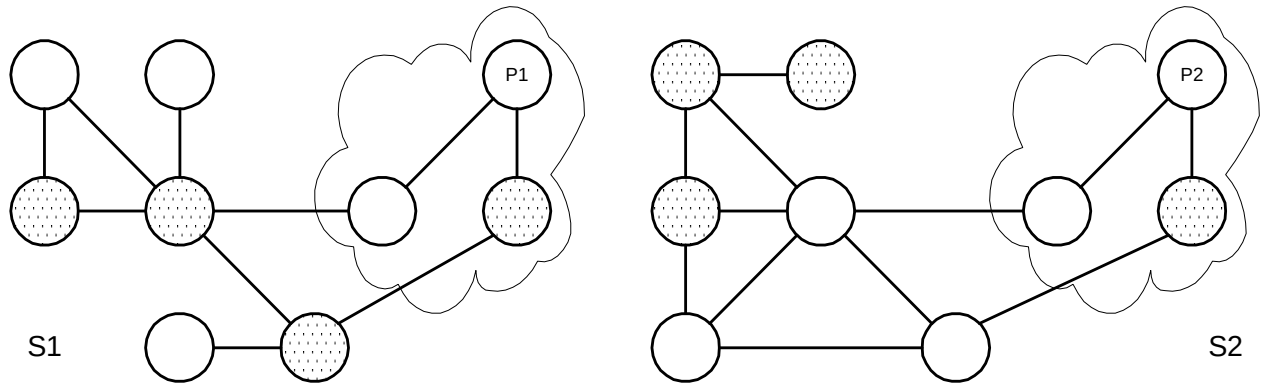

FIG. 3.2 – Les détecteurs de P_1 et P_2 donnent la même réponse car ils ont la même vue.

Cependant, nous faisons l'hypothèse que la réponse d'un oracle à un processeur est *uniquement déterminée* par l'*histoire* du processeur appelant. Plus précisément, la réponse de l'oracle est *vrai* si et seulement si un prédicat sur l'histoire du processeur appelant (induit par la spécification de la tâche à résoudre) est également vrai.

Définition 3.3 (Détecteur sur un Processeur) *Le* détecteur de défaillances du processeur P_i *est un oracle qui peut être questionné par* P_i*, que nous dénotons* $\mathcal{FD}_d^s(P_i)$*, et dont la réponse est uniquement déterminée par l'histoire locale* $\mathcal{V}_i^d[1..s]$ *de* P_i*.*

Le résultat de $\mathcal{FD}_d^s(P_i)$ peut être utilisé dans n'importe quelle configuration c_j d'une exécution. Le résultat de l'oracle $\mathcal{FD}_d^s(P_i)$ dans la configuration c_j est dénoté par $\mathcal{FD}_d^s(P_i, c_j)$.

Définition 3.4 (Détecteur dans une Configuration) *Le résultat réparti d'un* détecteur de défaillances dans une configuration c_j*, dénoté par* $\mathcal{FD}_d^s(c_j)$ *est la conjonction des résultats* $\mathcal{FD}_d^s(P_i, c_j)$ *de chacun des processeurs* P_i *présents dans le système.*

Intuitivement, si dans une configuration abstraite du système, la spécification est vérifiée, alors tout appel à un oracle par un processeur dans cette configuration doit renvoyer *vrai*. Dans le cas où la configuration abstraite ne vérifie pas la spécification, *au moins* un des détecteurs du système doit renvoyer *faux* au processeur qui le questionne.

De nombreux résultats de ce chapitre utilisent le fait que la réponse du détecteur est uniquement fonction de l'histoire du processeur appelant. Par exemple, considérons les deux systèmes S_1 et S_2 présentés figure 3.2 et résolvant la même tâche. Ces deux systèmes présentent une différence dans la topologie du graphe de communication et dans la valeur des variables abstraites associées à chaque nœud. Supposons qu'il existe un processeur P_1 dans S_1 et un processeur P_2 dans S_2 tels que les histoires de P_1 et de P_2 soient identiques, à un renommage des composants du système près. Alors si P_1 et P_2 utilisent le même oracle, celui-ci donne la même réponse à P_1 et à P_2.

Classification des Tâches Une tâche est $(d, s)-$locale si et seulement si il existe un détecteur de la classe $\mathcal{FD}_d^s$ pour cette tâche (c'est à dire que ce détecteur ne se trompe jamais), mais que tout détecteur de la classe $\mathcal{FD}_{d-1}^s$ ou $\mathcal{FD}_d^{s-1}$ peut se tromper (et n'est donc pas un détecteur de défaillances pour cette tâche). Un détecteur ne se trompe pas si et seulement si il répond *vrai* quand la spécification est vérifiée et que le détecteur n'a pas été victime d'une défaillance et faux sinon. Un détecteur peut se tromper de deux manières : en répondant *vrai* alors que la spécification n'est pas vérifiée, ou en répondant *faux* alors que la spécification est vérifiée et que le détecteur n'a pas été perturbé.

Plus formellement, ceci revient à dire que les conditions suivantes sont vérifiées :

1. Pour toute configuration c du système $\mathcal{S}$ et pour tout $1 \leq i \leq n$, $\mathcal{FD}_d^s(P_i, c)$ répond *vrai* quand c est correcte (c'est à dire vérifie sa spécification), et il existe un indice $1 \leq i \leq n$ tel que $\mathcal{FD}_d^s(P_i, c)$ répond *faux* quand c est incorrecte (c'est à dire ne vérifie pas sa spécification).

2. Il existe une configuration c qui est correcte (c'est à dire qui vérifie sa spécification) et un indice k tel que $\mathcal{FD}_{d-1}^s(P_k, c)$ répond *faux* ou il existe une configuration incorrecte c' dans laquelle pour tout $1 \leq k \leq n$, $\mathcal{FD}_{d-1}^s(P_i, c')$ répond *vrai*.

3. Il existe une configuration e qui est correcte (c'est à dire qui vérifie sa spécification) et un indice k tel que $\mathcal{FD}_d^{s-1}(P_k, c)$ répond *faux* ou il existe une configuration incorrecte e' dans laquelle pour tout $1 \leq k \leq n$, $\mathcal{FD}_d^{s-1}(P_i, c')$ répond *vrai*.

Critère de Localité Il s'avère que les capacités de détection de défaillances sont liées à la taille des histoires considérées sur chacun des processeurs du système. Nous distinguons deux paramètres :

1. *Distance* — La distance d est associée à chacune des vues de l'histoire $\mathcal{V}_i^d[1..s]$, où d est situé entre 0 et $r + 1$, et r est le rayon du graphe de communication du système.

2. *Temps* — Le temps s est associé au nombre de vues de $\mathcal{V}_i^d[1..s]$.

Les deux critères de localité que nous considérons pour les tâches rendent compte de la facilité avec laquelle des défaillances transitoires peuvent être détectées. Plus la localité en distance ou en temps est petite, et moins les oracles utilisent d'information pour donner leur réponse.

Informellement, si le résultat de l'oracle sur un processeur P_i utilise une histoire où les vues ne comprennent que l'état de P_i, alors cela signifie que le prédicat évalué par l'oracle est *purement local* : il est indépendant des voisins de P_i. Dans le cas particulier où la distance d de chaque vue de l'histoire de P_i vaut 1, le prédicat évalué est *locale*. Enfin, si $d = r + 1$, le prédicat évalué est *global*.

3.4 Détecteurs de Défaillances pour les Tâches Statiques

Dans cette section, nous étudions les détecteurs de défaillances pour les tâches statiques (voir section 2.4.5), où les variables abstraites du système restent figées à partir d'un point de l'exécution abstraite. Comme les tâches statiques sont 0-histoire locales, dans la suite, nous supposons que la propriété de localité se réfère uniquement à la localité en distance. Par conséquent, nous utilisons $\mathcal{V}_i^d$ pour dénoter la vue du processeur P_i. De manière similaire, nous utilisons $\mathcal{FD}_d$ au lieu de $\mathcal{FD}_d^1$.

Nous présentons maintenant plusieurs tâches statiques, leurs spécifications, et identifions la distance minimale nécessaire pour qu'un détecteur de défaillances transitoires puisse exister.

3.4.1 Tâches Globales

Dans cette section, nous présentons une classe de tâches qui nécessitent qu'au moins un processeur ait une vue comprenant la totalité du système. Dans la section 3.3, nous avons supposé que les vues $\mathcal{V}_i^d$ contenaient le graphe de communication jusqu'à la distance d du nœud i. Il y a également deux hypothèses de base concernant les informations relatives au graphe de communication ainsi stocké :

Hypothèse 3.1 (Liens Inclus) *Les identifiants de liens sont inclus dans les vues $\mathcal{V}_i^d$. En d'autres termes, $\mathcal{V}_i^d$ contient l'information relative à l'identité des processeurs à distance $d+1$ qui sont connectés à chaque processeur à distance d de P_i.*

Hypothèse 3.2 (Liens non Inclus) *Les identifiants des liens* ne *sont* pas *stockés dans les vues $\mathcal{V}_i^d$. En d'autres termes, $\mathcal{V}_i^d$ ne contient pas d'information sur les identités des processeurs à distance $d+1$.*

Ces hypothèses influent sur les conclusions qu'il est possible de tirer de la vue à distance d d'un processeur P_i :

1. Si l'hypothèse 3.1 est vraie, alors un centre du graphe de communication est capable de déterminer si il est un centre en utilisant $\mathcal{V}_i^r$, où r est le rayon du système. Si chaque lien est connecté à exactement deux nœuds, P_i possède une vue complète du système et sait qu'il est un centre. Dans le cas contraire, P_i sait qu'il n'a pas une vue complète du système.
2. Si c'est l'hypothèse 3.2 qui est vraie, le nœud i peut ne pas savoir qu'il est un centre en utilisant seulement $\mathcal{V}_i^r$, car il ne peut distinguer les deux cas suivants :
   (a) le rayon du réseau est r et i possède une connaissance complète du réseau ;
   (b) le rayon du réseau est $r+1$ et il manque à la vue de i au moins un nœud.

Notons toutefois que si la vue de P_i est $\mathcal{V}_i^{r+1}$, où r est le rayon du système, P_i est capable de déterminer si c'est un centre en vérifiant si sa vue contient des processeurs à distance $r+1$ de lui-même.

Nous sommes maintenant en mesure de considérer la première tâche, celle qui consiste à élire un chef dans un réseau.

Spécification de la tâche d'Election d'un chef

A chaque processeur P_i est associée une variable abstraite booléenne $\mathcal{L}_i$ qui vaut *vrai* si le nœud est élu et *faux* sinon. Il y a exactement un processeur P_l dont la variable abstraite $\mathcal{L}_l$ vaut *vrai.*

Lemme 3.1 *Si l'hypothèse 3.1 est vraie, la tâche d'élection d'un chef est r-distance locale.*

Preuve: La preuve consiste en un résultat d'impossibilité d'existence d'un détecteur de défaillances pour cette tâche qui soit dans $\mathcal{FD}_{r-1}$ et en la présentation d'un détecteur pour cette tâche qui soit dans l'ensemble $\mathcal{FD}_r$.

Soit c un centre du graphe. Par définition, il existe x tel que $x \in Boule(c,r)$ et $x \notin Boule(c,r-1)$. Soit y un nœud tel que $Dist(x,y) = D$, où D est le diamètre du graphe de communication.

Considérons les trois configurations suivantes :

1. c_1 dans laquelle x est élu et tous les autre nœuds ne le sont pas ;
2. c_2 dans laquelle y est élu et tous les autres nœuds ne le sont pas ;
3. c_3 dans laquelle aucun nœud n'est élu.

Supposons qu'il existe un détecteur pour la tâche de l'élection d'un chef qui soit dans $\mathcal{FD}_{r-1}$. Le résultat de ce détecteur dans les configurations $c_1$et c_2 doit être *vrai* sur chacun des processeurs, puisque chacune de ces configurations satisfait la spécification de l'élection d'un chef. D'un autre coté, dans la configuration c_3, au moins un détecteur doit répondre *faux.*

Considérons un nœud v tel que $x \notin Boule(v,r-1)$, $\mathcal{FD}_{r-1}(P_v, c_3)$ doit répondre *vrai* car il a la même vue que dans la configuration c_1. De plus, nous avons pour tout nœud u tel que $x \in Boule(u,r-1)$, $y \notin Boule(u,r-1)$. Dans le cas contraire, où $x \in Boule(u,r-1)$ et $y \in Boule(u,r-1)$, nous aurions :

$$\begin{array}{lcl} Dist(x,u) + Dist(y,u) & \leq & 2r-2 \\ Dist(x,y) & < & 2r-1 \\ & < & D \end{array}$$

mais nous avons choisi x et y tels que $Dist(x,y) = D$. Donc le résultat du détecteur $\mathcal{FD}_{r-1}(P_u, c_3)$ doit être *vrai* sur chaque nœud u pour lequel $x \in Boule(u,r-1)$ puisque la vue de u est identique à sa vue dans la configuration c_2. Par conséquent, le détecteur de $\mathcal{FD}_{r-1}$ répond *vrai* dans la configuration c_3 dans laquelle pourtant aucun chef n'est élu. Ceci achève la preuve d'impossibilité de l'existence d'un détecteur qui soit $(r-1)$-distance local.

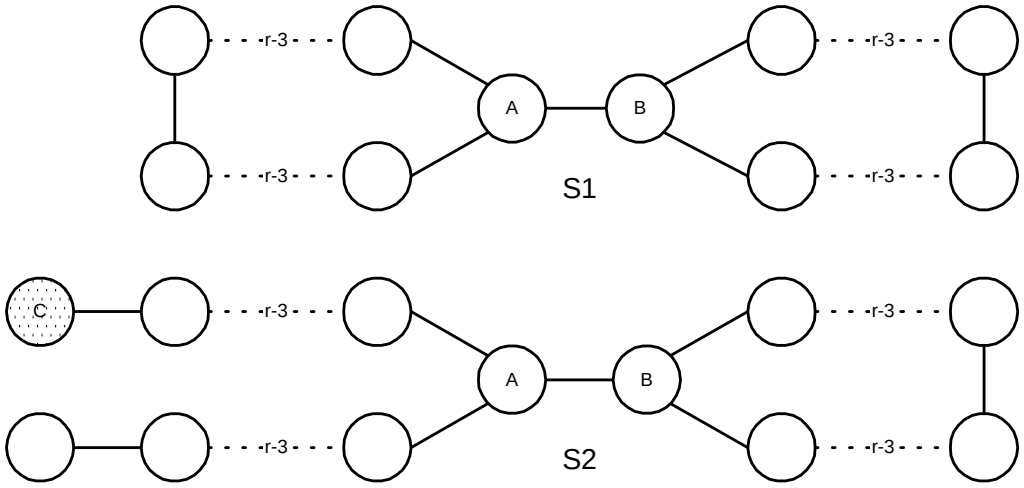

FIG. 3.3 – A et B sont centres dans S_1 mais B n'est pas centre dans S_2.

Présentons maintenant un détecteur de défaillances transitoires pour la tâche de l'élection d'un chef qui soit dans l'ensemble $\mathcal{FD}_r$. Avec une vue $\mathcal{V}_i^r$, et suivant l'hypothèse 3.1, P_i peut déterminer s'il connaît le système tout entier, c'est à dire si c'est un centre du graphe de communication. Construisons maintenant le détecteur de P_i :

1. si P_i n'est pas centre, $\mathcal{FD}_r(P_i)$ répond *vrai* (il ne détecte pas de faute) ;
2. si P_i est un centre, $\mathcal{FD}_r(P_i)$ vérifie au moyen de sa vue $\mathcal{V}_i^r$ qu'il existe un unique processeur P_l avec $\mathcal{L}_l = vrai$. Le détecteur signale une faute si et seulement si la condition précédente est fausse.

□

Ensuite nous présentons une preuve similaire pour le cas où la vue n'inclut pas les identités des liens de communication auxquels les processeurs sont connectés, c'est à dire que nous supposons l'hypothèse 3.2 vérifiée.

Lemme 3.2 *Si l'hypothèse 3.2 est vérifiée, la tâche d'élection d'un chef est* $(r+1)-$*distance locale.*

Preuve: La preuve consiste en un résultat d'impossibilité d'existence d'un détecteur de défaillances pour cette tâche qui soit dans $\mathcal{FD}_r$ et en la présentation d'un détecteur pour cette tâche qui soit dans l'ensemble $\mathcal{FD}_{r+1}$.

Considérons le système $\mathcal{S}_1$ de rayon r et constitué de $(4r-2)$ processeurs représenté dans la figure 3.3. Soit une configuration de ce système dans laquelle toutes les variables $\mathcal{L}_i$ sont à *faux*. Au moins un détecteur doit signaler une faute et, comme tous les autres processeurs n'ont qu'une vue partielle du système, seuls les détecteurs des nœuds A et B peuvent détecter cette faute.

Considérons maintenant un second système $\mathcal{S}_2$, contenant $4r$ processeurs et de même rayon r, représenté figure 3.3. Dans $\mathcal{S}_2$, toutes les variables $\mathcal{L}_i$ sont à *faux*, excepté la variable de C, qui est à *vrai* (de telle manière qu'un unique chef est élu). Dans les deux systèmes, les vues à distance r de B sont identiques. Donc dans les deux systèmes, le détecteur de B retourne la même réponse. Puisque le système $\mathcal{S}_2$ est correct, le détecteur de B ne détecte pas d'erreur dans $\mathcal{S}_2$, donc il n'en détecte pas non plus dans $\mathcal{S}_1$. En considérant un système dual à $\mathcal{S}_2$ où A ne serait pas centre, on peut conclure que dans $\mathcal{S}_1$, le détecteur de A est incapable de détecter une erreur.

Ceci induit une contradiction puisque dans le système $\mathcal{S}_1$, aucun détecteur ne détecte de faute. Par conséquent, il n'existe pas de détecteur dans l'ensemble $\mathcal{FD}_r$ pour la tâche d'élection d'un chef.

Présentons maintenant un détecteur de l'ensemble $\mathcal{FD}_{r+1}$ pour la tâche d'élection d'un chef. Avec une vue à distance $r+1$, un processeur P_i est en mesure de vérifier s'il connaît le système tout entier, c'est à dire s'il est un centre du graphe de communication. Construisons maintenant un détecteur pour P_i :

1. si P_i, n'est pas centre, le détecteur renvoie *vrai*.
2. si P_i est un centre, le détecteur vérifie à l'aide de sa vue à distance $(r+1)$ s'il existe exactement un processeur P_l dont la variable $\mathcal{L}_l$ vaut *vrai*. Le détecteur de P_i renvoie *faux* si et seulement si la condition précédente est fausse.

□

Remarque 3.1 *Pour toute tâche statique dont la spécification est un prédicat sur les configurations abstraites du système, il existe un détecteur dans l'ensemble $\mathcal{FD}_{r+1}$ (respectivement $\mathcal{FD}_r$) pour cette tâche si l'hypothèse 3.2 (respectivement l'hypothèse 3.1) est vérifiée. Plus précisément, ce détecteur utilise les centres pour vérifier si le prédicat global est vrai dans leur $(r+1)$-vues (respectivement $r-$vues).*

Dans la suite du chapitre, nous supposons que l'hypothèse 3.2 est vérifiée.

Spécification de la Tâche du Calcul du Nombre de Nœuds

Chaque processeur P_i est associé à une variable abstraite $Nombre_i$ qui contient le nombre de nœuds présents dans le système.

Lemme 3.3 *La tâche du calcul du nombre de nœuds est $(r+1)-$distance locale.*

Preuve: En accord avec la remarque 3.1, il suffit de montrer qu'il n'existe pas de détecteur dans l'ensemble $\mathcal{FD}_r$ pour la tâche du calcul du nombre de nœuds. Supposons le contraire et considérons le système $\mathcal{S}_1$ de la figure 3.3, avec $(4r-2)$ processeurs et de rayon r, dans lequel chaque processeur P_i a sa variable $Nombre_i$ à la valeur $4r$, qui est le nombre de processeurs du système $\mathcal{S}_2$ de la figure 3.3. Chaque processeur P_i différent de A et B (c'est à dire non-centre) possède moins de $(4r-4)$ processeurs dans sa vue à distance r.

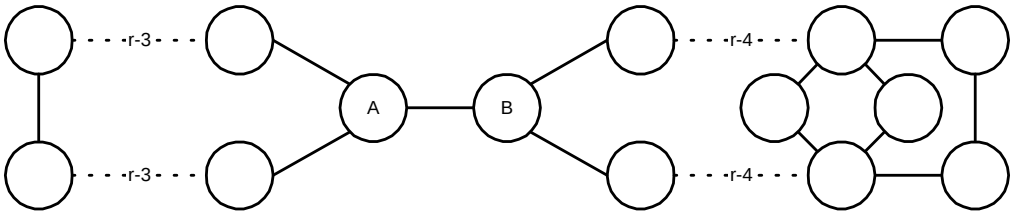

FIG. 3.4 – A et B sont centres.

Il est alors facile de construire un système $\mathcal{S}_1(i)$ doté de $4r$ processeurs pour chaque P_i du système $\mathcal{S}_2$, et tel que au moins un processeur possède la même vue à distance r que P_i. Par exemple, voir la figure 3.4 pour les nœuds à gauche de A dans $\mathcal{S}_1$. Par suite, le détecteur de P_i ne peut pas signaler de faute puisqu'il n'en a pas signalé dans $\mathcal{S}_1$ et qu'il a la même vue dans $\mathcal{S}_1(i)$. Par conséquent, seuls les détecteurs de A et B sont en mesure de détecter une faute dans $\mathcal{S}_1$. Mais A possède le même voisinage à distance r dans les systèmes $\mathcal{S}_1$ et $\mathcal{S}_2$. Donc il ne peut détecter de faute ni dans $\mathcal{S}_1$ ni dans $\mathcal{S}_2$. Pour des raisons similaires, B ne peut pas non plus détecter de faute, d'où une contradiction. □

Spécification de la tâche de Détermination d'un Centre

Chaque processeur P_i est associé à une variable abstraite booléenne $\mathcal{I}_i$ qui vaut *vrai* si et seulement si le processeur est un centre du graphe de communication.

Lemme 3.4 *La tâche de détermination d'un centre est $r-$distance locale.*

Preuve: La preuve qu'il n'existe aucun détecteur dans l'ensemble $\mathcal{FD}_{r-1}$ pour la tâche de détermination d'un centre se fait par la donnée de deux systèmes $\mathcal{S}_1$ et $\mathcal{S}_2$ décrits dans la figure 3.5, où un processeur P_i est dessiné en grisé quand sa variable $\mathcal{I}_i$ vaut *vrai*.

Le système $\mathcal{S}_1$ est correct, car les centres y sont correctement identifiées, donc chaque détecteur répond *vrai* dans ce système. Le système $\mathcal{S}_2$ est incorrect, car le processeur E y est incorrectement identifié comme un non-centre.

Chaque processeur situé dans le branche N du graphe de communication possède la même vue dans $\mathcal{S}_1$ et dans $\mathcal{S}_2$, donc chaque détecteur sur ces nœuds répond *vrai*. De plus, chaque processeur situé sur la branche W dans $\mathcal{S}_2$ possède la même vue que son homologue dans la branche N, donc chaque détecteur situé sur ces nœuds répond *vrai*. Enfin, chaque processeur de la branche E de $\mathcal{S}_2$ possède la même vue dans les systèmes $\mathcal{S}_1$ et $\mathcal{S}_2$, donc chaque détecteur situé sur un de ces nœuds répond *vrai*. En conséquence, aucune faute n'est détectée dans le système $\mathcal{S}_2$, pourtant incorrect.

Construisons maintenant un détecteur de l'ensemble $\mathcal{FD}_r$ pour la tâche de détermination d'un centre comme suit sur chaque processeur P_i :

1. si $\mathcal{I}_i = vrai$ (P_i prétend qu'il est un centre) alors le détecteur de P_i répond *faux* si et seulement si une des deux conditions suivantes est vérifiée :

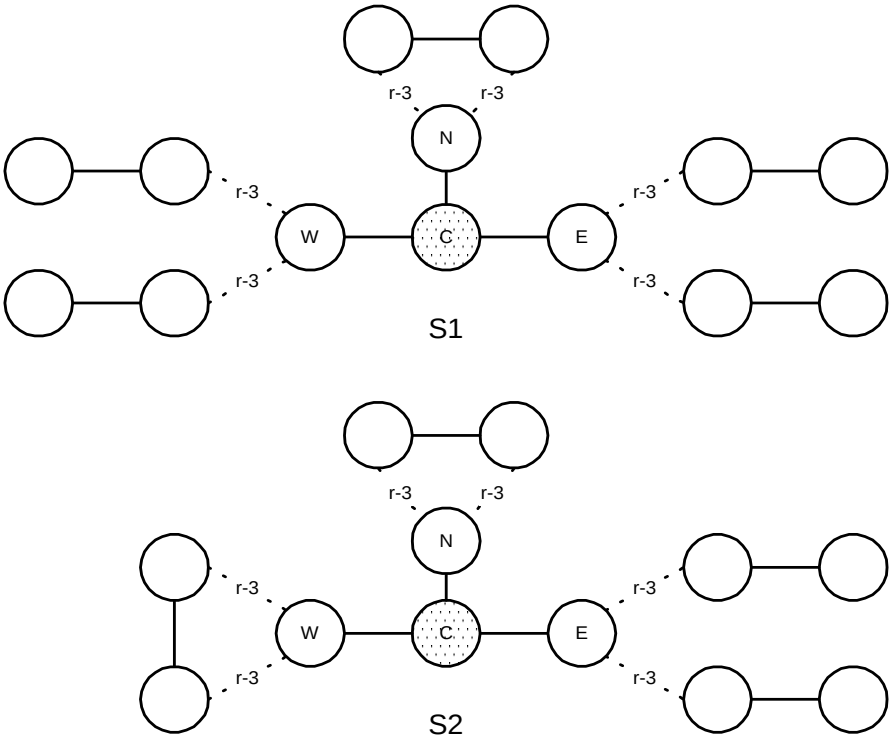

FIG. 3.5 – C est l'unique centre de S_1, mais C et E sont centres dans S_2.

(a) $\mathcal{V}_i^r$ contient un processeur P_j qui est un centre dans $\mathcal{V}_i^r$ et tel que $\mathcal{I}_j = \mathit{faux}$,

(b) $\mathcal{V}_i^r$ contient un processeur P_j qui n'est pas un centre dans $\mathcal{V}_i^r$ et tel que $\mathcal{I}_j = \mathit{vrai}$.

2. si $\mathcal{I}_i = \mathit{faux}$ (P_i prétend qu'il n'est pas un centre) alors le détecteur de P_i répond *faux* si et seulement si une des deux conditions suivantes est vérifiée :

(a) $\mathcal{V}_i^r$ contient uniquement des processeurs P_k tels que $\mathcal{I}_k = \mathit{faux}$,

(b) $\mathcal{V}_i^r$ contient deux processeurs P_j et P_k tels que $\mathcal{I}_j = \mathit{vrai}$ et $Dist(j,k) \geq r+1$ dans $\mathcal{V}_i^r$.

Dans le cas où le système est correct, tout centre P_i a sa variable $\mathcal{I}_i$ à *vrai* et tout non-centre P_j a sa variable $\mathcal{I}_j$ à *faux*. Alors, aucun détecteur situé sur un centre ne détecte d'inconsistance dans sa vue (qui couvre le système tout entier). Tout non-centre est à distance au plus r d'un centre (par définition d'un centre) dans chaque vue, donc aucun détecteur situé sur un non-centre ne retourne *faux*.

Si le système est incorrect, cela signifie que soit il existe un non-centre P_j tel que $\mathcal{I}_j = \mathit{vrai}$, soit un existe un centre P_k tel que $\mathcal{I}_k = \mathit{faux}$. Ceci nous amène à trois types de configurations :

1. Il existe un centre P_k tel que $\mathcal{I}_k = \mathit{vrai}$: alors le détecteur P_k est en mesure de détecter une erreur puisque sa vue couvre tout le graphe de communication.
2. Il n'existe aucun nœud P_i tel que $\mathcal{I}_i = \mathit{vrai}$, donc tous les détecteurs répondent *faux*.
3. Pour tout centre P_k, $\mathcal{I}_k = \mathit{faux}$, et il existe un non-centre P_j tel que $\mathcal{I}_j = \mathit{vrai}$. Alors P_k possède le réseau tout entier dans sa vue. Par définition d'un non-centre, il existe un processeur P_i du réseau tel que $Dist(j,i) \geq r+1$. Donc P_k possède dans sa vue deux nœuds à distance strictement supérieure à r et l'un d'entre eux prétend qu'il est un centre, par conséquent le détecteur de P_k détecte une erreur.

□

3.4.2 Tâches Locales

Spécification de la tâche de Construction d'un Ensemble Maximal Indépendant

A chaque processeur P_i est associée une variable abstraite booléenne $\mathcal{IS}_i$. Il n'existe aucune paire de voisins telle que leur variable soit à *vrai* simultanément. De plus, chaque processeur P_i dont $\mathcal{IS}_i = \mathit{faux}$ possède au moins un voisin P_j tel que $\mathcal{IS}_j = \mathit{vrai}$.

Lemme 3.5 *La tâche de construction d'un ensemble maximum indépendant est* $1-$*distance locale.*

Preuve: Nous montrons l'impossibilité d'un détecteur pour cette tâche dans l'ensemble $\mathcal{FD}_0$ et construisons un détecteur dans l'ensemble $\mathcal{FD}_1$.

Avec une vue à distance 1, un processeur connaît la valeur $\mathcal{IS}_j$ de chacun de ses voisins. Un détecteur du processeur P_i peut alors vérifier que si $\mathcal{IS}_i = \mathit{vrai}$, alors $\forall j \in \Gamma_G^1(i)$, $\mathcal{IS}_i \neq \mathcal{IS}_j$, et que si $\mathcal{IS}_i = \mathit{faux}$, alors $\exists j \in \Gamma_G^1(i)$, $\mathcal{IS}_i \neq \mathcal{IS}_j$. Le détecteur de P_i indique

l'occurence d'une faute si l'une des propriétés précédentes n'est pas vérifiée. Le test précédent assure que les valeurs des variables $\mathcal{IS}$ construisent un ensemble maximal indépendant.

Par définition des détecteurs 0-distance locaux, seules les variables abstraites propres à un processeur sont incluses dans sa vue. Par conséquent, aucune faute ne peut être détectée dans une configuration où toutes les variables $\mathcal{IS}_i$ sont à *vrai*. □

Une preuve similaire pour la tâche de coloration peut être établie.

Spécification de la tâche de Coloration de Graphe

Chaque processeur P_i est associé à une variable abstraite $\mathcal{C}_i$ qui représente sa couleur. De plus, deux voisins P_i et P_j ne peuvent avoir la même couleur.

Lemme 3.6 *La tâche de coloration de graphe est* 1*-distance locale.*

Preuve: La preuve se fait par la construction d'un détecteur dans l'ensemble $\mathcal{FD}_1$ pour cette tâche et l'impossibilité d'existence d'un détecteur dans l'ensemble $\mathcal{FD}_0$.

Avec une vue à distance 1, le détecteur de P_i est en mesure de vérifier que $\forall j \in \Gamma^1_{\mathcal{G}}(i), \mathcal{C}_i \neq \mathcal{C}_j$. Il indique l'occurence d'une faute si la propriété précédente n'est pas vérifiée.

Par définition des détecteurs 0-distance locaux, seules les variables abstraites propres à un processeur sont incluses dans sa vue. Par conséquent, aucune faute ne peut être détectée dans une configuration où toutes les variables $\mathcal{C}_i$ sont semblables. □

Spécification de la tâche de Mise à Jour de Topologie

Chaque processeur P_i est associé à une variable abstraite $\mathcal{T}_i$, qui contient la représentation du graphe de communication, par exemple au moyen d'une matrice d'adjacence ou bien de la liste des arcs du graphe de communication.

Lemme 3.7 *La tâche de mise à jour de topologie est* $1-$*distance locale.*

Preuve: La preuve se fait par la construction d'un détecteur dans l'ensemble $\mathcal{FD}_1$ pour cette tâche et l'impossibilité d'existence d'un détecteur dans l'ensemble $\mathcal{FD}_0$.

Avec une vue à distance 1, le détecteur de P_i est en mesure de vérifier que $\mathcal{T}_i = \mathcal{T}_j$ pour chaque voisin P_j. Le détecteur de P_i est capable de détecter l'occurence d'une erreur dans le cas où il existe un voisin pour lequel l'égalité précédente n'est pas vérifiée. De plus, le détecteur de P_i vérifie si la topologie locale de P_i apparaît correctement dans $\mathcal{T}_i$. Ce test assure que les valeurs identiques communes des variables $\mathcal{T}$ soient correctes, puisque chaque processeur a identifié sa topologie locale dans $\mathcal{T}$.

Par définition des détecteurs 0-distance locaux, seules les variables abstraites propres à un processeur sont incluses dans sa vue. Par conséquent, aucune faute ne peut être détectée dans une configuration où toutes les variables $\mathcal{T}_i$ de chaque processeur P_i possède la topologie locale de P_i, c'est à dire P_i et ses voisins, sans le reste (non vide) du système. □

3.4.3 Autres Tâches Statiques

Dans cette section, nous présentons la tâche de construction d'un arbre enraciné, qui est $\lceil n/4 \rceil-$distance locale.

Spécification de la tâche de Construction d'Arbre Enraciné

A chaque processeur P_i est associée une variable abstraite $\mathcal{P}_i$ contenant un pointeur vers l'un de ses voisins, choisi pour être son parent dans l'arbre. Un unique processeur du système, P_r appelé la *racine*, possède une valeur spécifique *nul* dans sa variable abstraite associée $\mathcal{P}_r$, tandis que la valeur des variables abstraites associées aux autres processeurs du système définit un arbre enraciné en P_r.

Lemme 3.8 *La tâche de construction d'arbre enraciné est $\lceil \frac{n}{4} \rceil-$distance locale.*

Preuve: La preuve se fait en montrant l'impossibilité de l'existence d'un détecteur pour cette tâche dans l'ensemble $\mathcal{FD}_{\lceil \frac{n}{4} \rceil -1}$ et par la construction d'un détecteur pour cette tâche qui soit dans l'ensemble $\mathcal{FD}_{\lceil \frac{n}{4} \rceil}$.

Le détecteur de défaillances sur chaque processeur non-racine vérifie s'il existe un cycle ou bien s'il existe une preuve que l'arbre connecté à la racine n'inclut pas au moins un processeur. Le détecteur de la racine vérifie si le sous-arbre qui lui est connecté comprend bien tous les processeurs. Il est clair que lorsque le système encode un arbre enraciné sur P_r, aucune erreur n'est détectée. Si un processeur non-racine n'a pas de parent, cela est détecté immédiatement. Il ne reste que le cas où seul un sous-arbre est enraciné sur la racine et où tous les autres processeurs forment un cycle.

Un détecteur de défaillances qui utilise des vues de rayon $\lceil \frac{n}{4} \rceil$ peut détecter des cycles de rayon au plus $2 \times \lceil \frac{n}{4} \rceil$ processeurs. Par conséquent, un cycle regroupant $2 \times \lceil \frac{n}{4} \rceil + 1$ processeurs ne peut être détecté. Considérons un graphe avec un tel cycle et un sous-arbre enraciné sur P_r. Ce sous-arbre contient au plus $n - 2 \times \lceil \frac{n}{4} \rceil - 1$ nœuds, donc son diamètre est au plus $n - 2 \times \lceil \frac{n}{4} \rceil - 2$. Soit B l'ensemble des nœuds qui sont dans le sous-arbre connecté à P_r ou à des voisins directs des nœuds de ce sous-arbre. Le diamètre de la partie du graphe de communication qui connecte les nœuds de B (le graphe dont seules des arêtes connectant des nœuds de B apparaîssent) est au plus $n - 2 \times \lceil \frac{n}{4} \rceil$. Si un centre P_c de l'ensemble B possède une vue à distance $\lceil \frac{n-2\times\lceil \frac{n}{4} \rceil}{2} \rceil$, il est en mesure de voir tous les nœuds de B. Puisque $\lceil \frac{n-2\times\lceil \frac{n}{4} \rceil}{2} \rceil \leq \lceil \frac{n}{4} \rceil$, tous les nœuds de B sont inclus dans la vue de P_c.

Maintenant, si *tous* les nœuds qui sont des voisins directs d'un processeur du sous-arbre (mais pas inclus eux-mêmes dans le sous-arbre) n'ont pas choisi leur parent parmi les nœuds du sous-arbre, alors le centre est en mesure de détecter une faute de manière appropriée, puisqu'il détecte des nœuds qui ne sont pas connectés à la racine.

Afin de prouver que la tâche n'est pas $(\lceil \frac{n}{4} \rceil - 1)$-local, nous considérons une configuration du système c dans laquelle une chaîne de $\lceil n/2 \rceil$ processeurs sont connectés à la racine et le reste des processeurs forme un cycle. Dans une telle configuration, certains processeurs

doivent détecter une faute. Nous montrons que chacun de ces processeurs détecte également une faute dans une configuration qui encode un arbre enraciné (et donc correcte).

Supposons que le détecteur d'un processeur P_i contenu dans le cycle identifie une faute dans la configuration c. Alors il existe au moins un processeur P_j dans le cycle qui ne soit pas inclus dans $\mathcal{V}_i^d$ (où $d = (\lceil \frac{n}{4} \rceil - 1)$). Il existe une configuration c', qui inclus une arête supplémentaire de P_j à la racine, et où P_j peut avoir choisi cette arête comme arête d'arbre de telle manière qu'aucun cycle n'existe dans c'. Cependant, P_i possède la même vue $\mathcal{V}_i^d$ dans c' et par conséquent son oracle détecte une faute dans c'.

Si le détecteur qui signale la faute est appelé par un processeur placé sur la chaîne reliée à la racine, alors il existe au moins un processeur P_j, sur la chaîne et un processeur P_k sur le cycle qui n'est pas inclus dans $\mathcal{V}_i^d$. Donc, le détecteur de P_i indique une faute lorsque le système est dans une autre configuration, dans laquelle il n'y a pas de cycle puisque P_j et P_k sont connectés par une arête et que P_j choisit P_k comme son parent. □

Spécification de la tâche de Construction d'un Circuit Hamiltonien (dans un système Hamiltonien)

Chaque processeur P_i distingue une arête entrante ($in(i)$) et une arête sortante ($out(i)$), qui induisent globalement un circuit Hamiltonien.

Lemme 3.9 *La tâche de construction d'un circuit Hamiltonien est $\lceil n/4 \rceil$−distance locale.*

Preuve: Notons tout d'abord qu'un appelé par un processeur qui ne possède pas exactement une arête entrante et une arête sortante peut détecter une faute. Si la structure induite par les variables $in(i)$ et $out(i)$ n'est pas un circuit Hamiltonien, il existe nécessairement plusieurs circuits couvrant le graphe de communication (chaque processeur est dans un unique circuit). Cependant, chacun de ces circuits comprennent moins de $n/2$ processeurs et chaque processeur du circuit possède le circuit tout entier dans sa vue à distance $\lceil n/4 \rceil$. Par conséquent les défaillances peuvent être détectées.

Maintenant, prouvons que cette tâche ne possède pas de détecteur dans l'ensemble $\mathcal{FD}_{\lceil n/4 \rceil - 1}$, et considérons les deux systèmes $\mathcal{S}_1$ et $\mathcal{S}_2$ présentés dans la figure 3.6.

A la fois dans $\mathcal{S}_1$ et $\mathcal{S}_2$, le processeur A à la même $\mathcal{V}^{\lceil n/4 \rceil - 1}$, par conséquent son détecteur ne peut pas détecter de faute dans $\mathcal{S}_1$. Puisque le système est symétrique, aucun détecteur ne peut détecter de faute dans $\mathcal{S}_1$, qui ne contient pourtant aucun circuit Hamiltonien. □

Spécification de la tâche de Construction d'un Circuit Eulérien (dans un système Eulérien)

Chaque processeur P_i possède $\delta(i)/2$ variables. Chaque variable l_j contient un couple d'indices d'arêtes dont une extrémité atteint i, de telle manière que ces arêtes apparaissent l'une après l'autre dans un circuit Eulérien.

Lemme 3.10 *La tâche de construction d'un circuit Eulérien est $\lceil n/4 \rceil$−distance locale.*

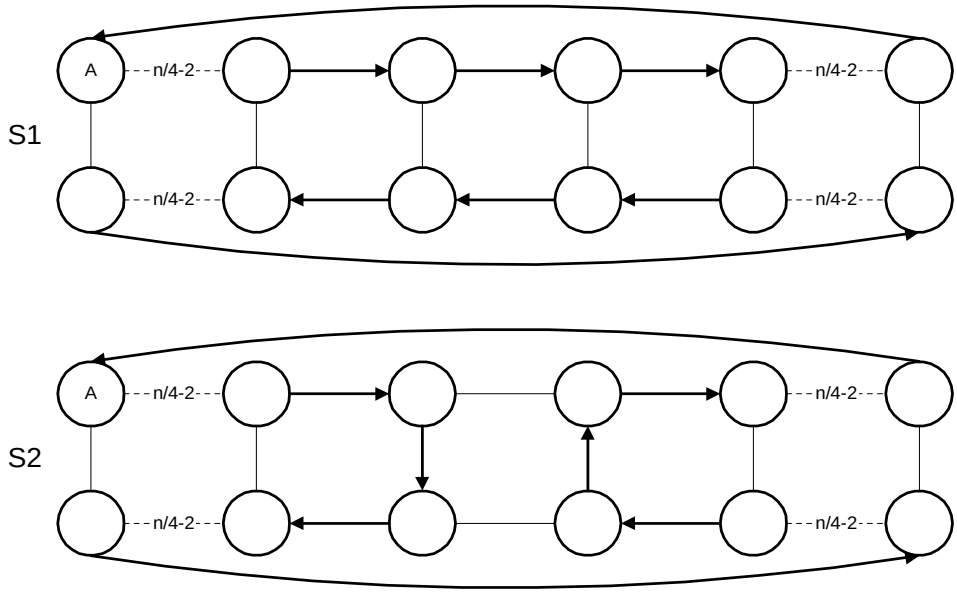

FIG. 3.6 – Le système S_1 est incorrect, mais le système S_2 est correct.

Preuve: Notons tout d'abord qu'un détecteur placé sur un processeur P_i ayant une arête apparaissant dans deux l_j distincts est en mesure de détecter une faute. Pour prouver qu'il y a exactement un circuit, il suffit de regarder à distance $\lceil n/4 \rceil$, comme dans la preuve du circuit Hamiltonien. Pour le résultat d'impossibilité d'un détecteur dans l'ensemble $\mathcal{FD}_{\lceil n/4 \rceil - 1}$, considérons les deux systèmes présentés dans la figure 3.7.

Les processeurs A et B ont le même $\mathcal{V}^{\lceil n/4 \rceil - 1}$ dans les deux systèmes, donc aucun de leurs détecteurs ne peut détecter de faute dans le premier. Comme le système est symétrique, aucun détecteur ne peut détecter de faute dans $\mathcal{S}_1$, où pourtant aucun circuit Eulérien n'existe. □

Maintenant montrons qu'il existe une classe de tâches qui requiert un détecteur de défaillance de rayon i (où i n'est pas lié à n) pour tout entier i.

Spécification de la tâche de Construction d'une x-Partition dans un Arbre

Chaque processeur P_i est associé à une variable abstraite booléenne $\mathcal{B}_{ij}$ pour chacune de ses arêtes incidentes (i, j). Si les arêtes pour lesquelles $\mathcal{B} = vrai$, qui sont appelées arêtes de *bordure,* sont déconnectées, alors l'arbre est partitionné en composantes connexes de diamètre au plus x, où x est un entier positif plus petit que n.

Lemme 3.11 *La tâche de construction d'une x-partition dans un arbre est $\lfloor \frac{x}{2} \rfloor$-distance locale.*

Preuve: La preuve consiste en l'impossibilité d'existence d'un détecteur dans l'ensemble $\mathcal{FD}_{\lfloor \frac{x}{2} \rfloor - 1}$ pour cette tâche et par la construction d'un détecteur dans l'ensemble $\mathcal{FD}_{\lfloor \frac{x}{2} \rfloor}$.

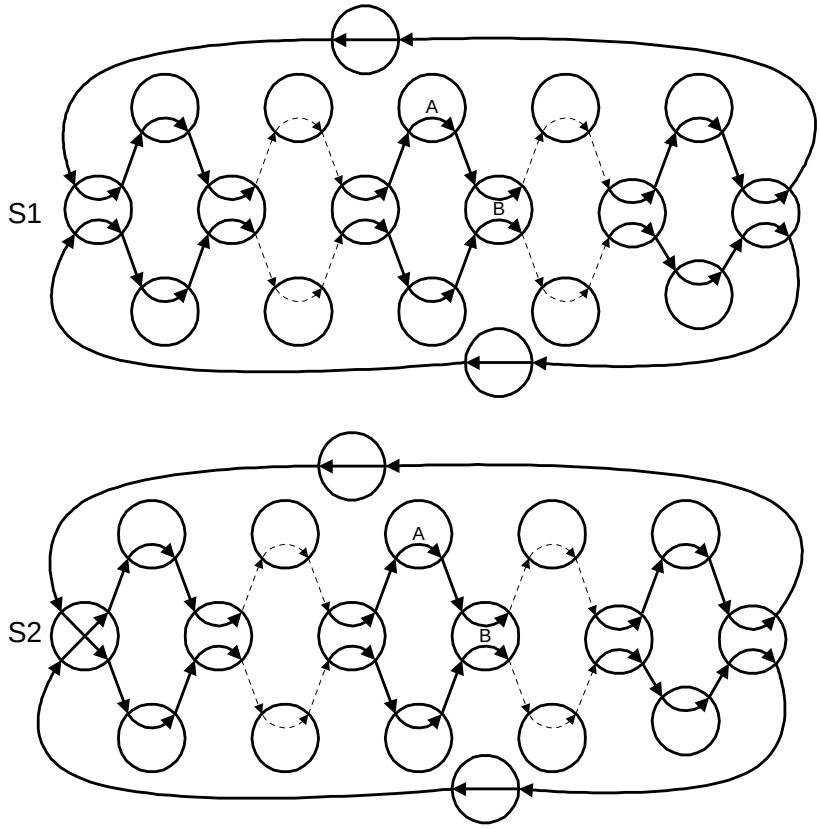

FIG. 3.7 – Le système S_1 est incorrect et le système S_2 est correct.

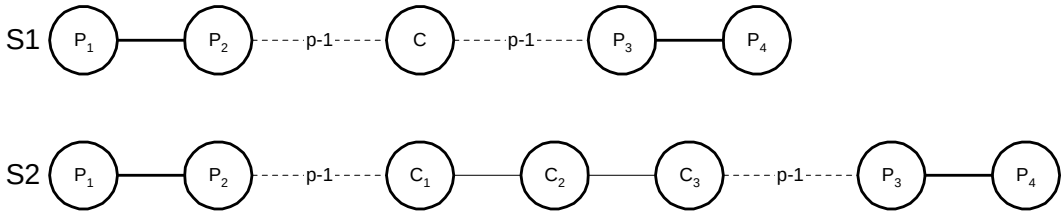

FIG. 3.8 – Le système S_1 est correct et le système S_2 est incorrect.

Le résultat d'impossibilité consiste à considérer deux systèmes $\mathcal{S}_1$ (qui est correct) et $\mathcal{S}_2$ (qui est incorrect) qui sont présentés dans la figure 3.8 et à montrer que si un détecteur répond *vrai* dans le premier système, alors il répond également *vrai* dans le deuxième système. Dans les deux figures, les arêtes en gras dénotent des arêtes de bordure.

Considérons un entier $p = \lfloor \frac{x}{2} \rfloor$ (soit $x = 2 \times p$ si x est pair et $x = 2 \times p + 1$ si x est impair). Supposons que dans $\mathcal{S}_1$, le processeur C possède une vue à distance $p - 1$. Alors la vue de C ne peut contenir les processeurs P_2 et P_3. Puisque le premier système est correct, aucun détecteur ne répond *faux*. Dans le système $\mathcal{S}_2$, les processeurs C_1, C_2 et C_3 possèdent la même vue que le processeur C dans $\mathcal{S}_1$. Évidement, tous les autres nœuds du système $\mathcal{S}_2$ possèdent la même vue que dans le système $\mathcal{S}_1$, donc aucun d'entre eux ne peut détecter d'erreur. Par conséquent, aucun détecteur ne signale d'erreur dans le système $\mathcal{S}_2$, bien qu'il soit incorrect.

Maintenant supposons que chaque détecteur dispose d'une vue à distance p. Si un processeur P_i possède dans sa vue une chaîne de longueur $2p$ qui ne contient aucune arête de bordure, alors son détecteur répond *faux*, et *vrai* dans le cas contraire. Effectivement, si une telle chaîne existe, cela signifie qu'il peut exister une composante connexe contenant cette chaîne et de diamètre $2 \times p + 2 > x$. Si aucune composante connexe n'est de diamètre supérieur à x, alors sur chaque chaîne de taille $2 \times p$, il existe au moins une arête de bordure. □

3.5 Détecteurs de Défaillances pour les Tâches Dynamiques

Dans cette section, nous considérons les tâches dynamiques. A la différence des sections précédentes où les détecteurs de défaillances s'appliquent aussi bien aux systèmes synchrones qu'asynchrones, nous nous restreignons ici aux détecteurs de défaillances pour des systèmes synchrones. Nous présentons des tâches qui sont s-histoire locales, avec $s > 1$. Ici, s définit la taille de l'histoire $\mathcal{V}_i^d[1..s]$ de chaque processeur P_i. Le système étant synchrone, chaque vue $\mathcal{V}_i^d[1..s]$ est liée à un instant différent. Ce tableau est désigné ci-après par le terme d'*histoire locale* du processeur P_i. Chacun des $\mathcal{V}_i^d$ est une vue des composants du système jusqu'à la distance d du processeur P_i.

Nous présentons maintenant plusieurs tâches dynamiques, leur spécification, puis déterminons la taille minimale de l'histoire locale requise pour qu'un détecteur de défaillances existe pour cette tâche.

3.5.1 Tâches Bornées

Nous débutons avec la tâche triviale du privilège borné. De toute évidence, cette tâche ne requiert aucune communication inter-processeurs.

Spécification de la tâche du Privilège Borné

A chaque processeur P_i est associée une variable abstraite booléenne $\mathcal{P}riv_i$. Pour tous les processeurs P_i, $\mathcal{P}riv_i$ vaut *vrai* exactement une fois (une autre variante est *au moins une fois*) lors de c étapes synchrones ($c \geq 2$).

Lemme 3.12 *La tâche du privilège borné est* $0-$*distance locale,* $c-$*histoire locale.*

Preuve: Une histoire locale de $c-1$ vues et telle que dans chaque vue la variable $\mathcal{P}riv_i$ vaut *faux* ne donne aucune indication quant à la violation de la spécification. D'un autre coté, il est évident qu'une histoire locale de c vues est suffisante pour détecter une défaillance. □

Spécification de la tâche du Dîner de Philosophes Borné

Chaque processeur P_i est associé à une variable abstraite booléenne $\mathcal{P}riv_i$. Pour tous les processeurs P_i, $\mathcal{P}riv_i$ vaut *vrai* au moins une fois toute les c étapes synchrones ($c \geq 2$). De plus, pour chaque couple de processeurs voisins P_i et P_j, si $\mathcal{P}riv_i = vrai$ alors $\mathcal{P}riv_j = faux$.

Lemme 3.13 *La tâche du dîner de philosophes borné est* $1-$*distance locale,* $c-$*histoire locale.*

Preuve: Tout d'abord, prouvons qu'il n'existe pas de détecteur de défaillances transitoire qui soit 0-distance local pour la tâche du dîner de philosophes borné. S'il existait un tel détecteur, il ne pourrait pas détecter que deux voisins P_i et P_j soient tels que $\mathcal{P}riv_i = \mathcal{P}riv_j = vrai$.

Puis nous prouvons qu'il n'existe pas de détecteur de défaillances transitoires qui soit $(c-1)$-histoire local pour la tâche du dîner de philosophes borné.

Si $c = 2$, un détecteur $(c-1)$-histoire local ne pourrait utiliser que la vue courante pour détecter une faute.

Considérons alors deux exécutions possibles e_1 et e_2 d'un système constitué de deux processeurs P_1 et P_2, comme indiqué dans la figure 3.9. Dans l'exécution e_1, $\mathcal{P}riv_1$ et $\mathcal{P}riv_2$ valent *vrai* alternativement toutes les deux unités de temps, donc cette exécution est correcte. Visiblement, l'exécution e_2 est incorrecte puisque $\mathcal{P}riv_2$ ne vaut jamais *vrai*. Dans toute configuration C_n de e_2, toute 1-histoire est égale à la 1-histoire dans la configuration $C_{2\times n}$ de e_1. Puisque le détecteur doit répondre *vrai* dans toute configuration de e_1, il doit également

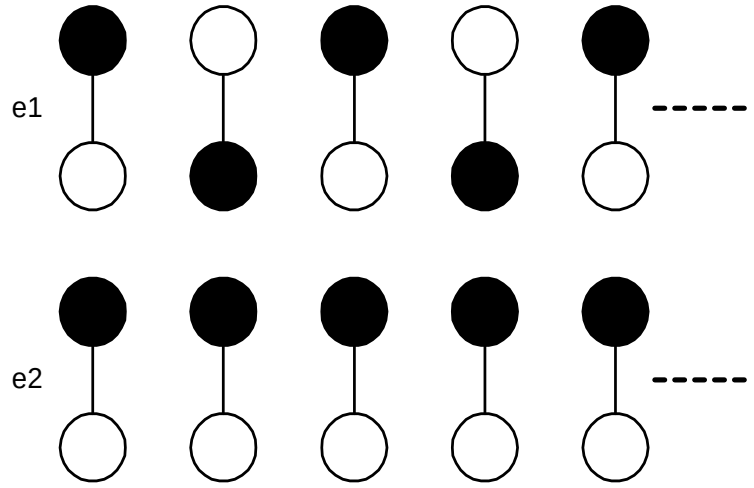

FIG. 3.9 – e_1 est une exécution correcte, mais e_2 est une exécution incorrecte.

répondre *vrai* dans toute configuration de $e2$, qui est une exécution incorrecte, donc ce détecteur est incapable de détecter une faute.

Si $c \geq 3$, alors considérons un système contenant deux processeurs P_1 et P_2 et considérons deux exécutions de ce système comme décrit dans la figure 3.10. Les exécutions e_1 et e_2 consistent en un facteur de répétition de taille c.

Dans chacune des exécution e_1 et e_2, les processeurs P_1 et P_2 ont leur variable $\mathcal{P}riv$ égale à *vrai* toutes les c configurations. De plus, dans chacune des configurations de ces exécutions, nous n'observons jamais $\mathcal{P}riv_1 = \mathcal{P}riv_2 = vrai$. Par conséquent, les exécutions e_1 et e_2 sont correctes, donc les détecteurs de défaillances sur P_1 et P_2 répondent *vrai* dans chacune des configurations de ces exécutions. Maintenant considérons l'exécution e_f du même système comme décrit dans la figure 3.10.

L'exécution e_f est constituée d'un facteur de répétition de taille $2 \times c$. Dans ce facteur, les premières c configurations sont les mêmes que dans l'exécution e_1, tandis que les c suivantes sont les mêmes que dans l'exécution e_2. Dans l'exécution e_f, $\mathcal{P}riv_1$ vaut *vrai* toutes les c configurations, mais $\mathcal{P}riv_2$ vaut *vrai* alternativement toutes les $c+1$ et toutes les $c-1$ configurations, donc l'exécution e_f est incorrecte.

Dans les configuration C_1 à C_{c+1} de l'exécution e_f, la $(c-1)$-histoire des processeurs P_1 et P_2 est la même que dans les configurations C_1 à C_{c+1} de l'exécution e_1, donc les deux détecteurs de défaillances répondent *vrai* dans ces configurations de e_f. Dans les configurations C_{c+2} à $C_{2\times c+1}$ de l'exécution e_f, la $(c-1)$-histoire des processeurs P_1 et P_2 est la même que dans les configurations C_{c+2} à $C_{2\times c+1}$ de l'exécution e_2, donc les deux détecteurs de défaillances répondent *vrai* dans ces configurations de e_f.

En utilisant le même raisonnement, les $(c-1)$-histoires des processeurs P_1 et P_2 dans les configurations $C_{(k-1)\times c+2}$ à $C_{k\times c+1}$ de l'exécution e_f sont les mêmes que celles de l'exécution e_1 si k est impair et les mêmes que celles de l'exécution e_2 si k est pair. Puisque les histoires

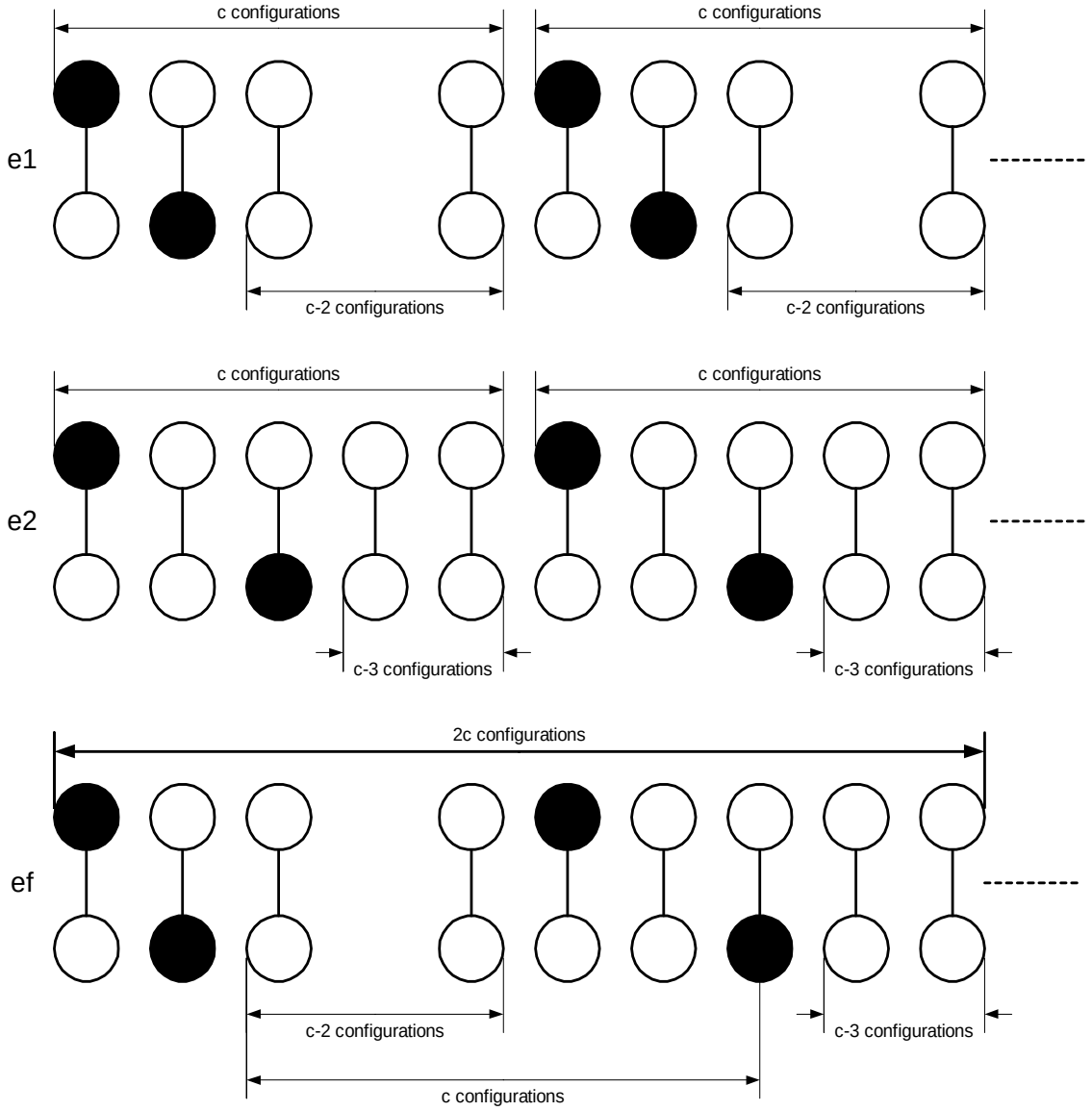

FIG. 3.10 – e_1 et e_2 sont des exécutions correctes, mais e_f est une exécution incorrecte.

dans l'exécution e_f pourraient être celles d'exécutions correctes, aucune faute n'est détectée dans aucune des configurations de l'exécution e_f, qui est pourtant une exécution incorrecte.

D'un autre coté, il est évident qu'une histoire locale de c vues est suffisante pour détecter une violation du premier prédicat de la spécification de la tâche. De surcroît, pour s'assurer que deux processeurs voisins ne sont pas simultanément privilégiés, une vue de diamètre 1 est nécessaire et suffisante. □

3.5.2 Tâches Déterministes et Non-interactives

Dans une exécution synchrone d'un protocole non-probabiliste, non-interactif et qui utilise un espace mémoire borné, certaines configurations abstraites doivent être atteintes plus d'une fois au cours d'une exécution, et par conséquent le système repasse par ces configurations infiniment souvent dans chacune des exécutions ifinies. Dès lors, il est possible d'identifier un *motif de répétition borné*, où les actions des processeurs se répètent indéfiniment.

Il est possible d'ajoindre à chaque processeur une histoire locale qui inclut toutes les vues du motif de répétition dans l'ordre où elles doivent être répétées. Notons que lorsque la distance des vues est égale au diamètre du système, le détecteur de défaillances que nous venons de présenter peut servir d'implantation de la tâche.

3.5.3 Tâches Equitables

Spécification de la tâche du Privilège Equitable

A chaque processeur P_i est associée une variable abstraite booléenne $\mathcal{P}riv_i$. Pour tous les processeurs P_i, $\mathcal{P}riv_i$ vaut *vrai* infiniment souvent dans chaque exécution synchrone.

Lemme 3.14 *Quels que soient les entiers d et l, il n'existe aucun détecteur de défaillances transitoires pour la tâche du privilège équitable qui soit d-distance local et l-histoire local.*

Preuve: Supposons qu'il existe deux entiers d et l tels qu'il existe un détecteur de défaillances transitoires pour la tâche du privilège équitable qui soit d-distance local et $l-$histoire local. Considérons un système où dans chaque exécution synchrone, tous les processeurs ont la même valeur de privilège (pour tout processeur P_i, et pour toute configuration c, toutes les variables $\mathcal{P}riv_i$ sont égales). Une exécution de ce système est donnée par les valeurs successives de $\mathcal{P}riv_i$, la variable d'un unique processeur P_i. Nous considérons maintenant deux exécutions R_1 (qui est correcte) et R_2 (qui est incorrecte) et prouvons que si le détecteur de défaillances transitoires $l-$histoire local répond *vrai* dans l'exécution R_1, il répond également *vrai* dans l'exécution R_2, et que par conséquent il ne détecte pas l'exécution incorrecte.

$$
\begin{aligned}
R_1 &= \underbrace{faux, \ldots, faux}_{l \text{ fois}}, vrai, vrai, \ldots \\
R_2 &= \underbrace{faux, \ldots, faux}_{l \text{ fois}}, faux, faux, \ldots
\end{aligned}
$$

Dans l'exécution R_1, la variable $\mathcal{P}riv_i$ vaut *vrai* infiniment souvent, donc cette exécution est correcte. Par suite, le détecteur de défaillances transitoires sur le processeur P_i doit répondre *vrai* dans chacune des configurations.

Dans l'exécution R_2, la variable $\mathcal{P}riv_i$ ne vaut jamais *vrai*, donc cette exécution est incorrecte. Par suite, le détecteur de défaillances transitoires sur le processeur P_i doit répondre *faux* dans au moins une configuration.

Les configurations c_0 à c_{l-1} sont identiques dans R_1 et dans R_2, donc les histoires locales $\mathcal{V}_i^0[1..l]$ sont égales dans ces configurations. Comme le détecteur de défaillances transitoires répond *vrai* dans les configurations $c_0 \dots c_{l-1}$ dans R_1 (qui est correcte), il répond également *vrai* dans les configurations $c_0 \dots c_{l-1}$ de R_2. Notons que dans la configuration c_{l-1} de R_1 ou R_2, la variable $\mathcal{V}_i^0[1..l]$ vaut $\underbrace{faux, \dots, faux}_{l \text{ fois}}$.

Maintenant considérons la configuration c_k de R_2, avec $k \geq l$. Dans la configuration c_k, $\mathcal{V}_i^0[1..l]$ vaut $\underbrace{faux, \dots, faux}_{l \text{ fois}}$, donc le détecteur de défaillances transitoires doit répondre *vrai* dans la configuration c_k.

Puisque dans toute configuration de R_2, le détecteur de défaillances transitoires $l-$histoire local répond *vrai*, il ne détecte aucune faute dans R_2, qui est un exécution incorrecte. □

3.6 Détecteurs de Défaillances pour les Algorithmes

A la différence des détecteurs de défaillances pour des tâches, les détecteurs de défaillances pour des algorithmes (c'est à dire l'implantation d'une tâche) ont la possibilité d'utiliser l'état tout entier d'un processeur (et plus seulement l'état des variables abstraites définies pour une tâche particulière). Ceci permet de réduire considérablement la quantité de mémoire nécessaire à la détection d'une défaillance, comme dans l'exemple ci-dessous :

Exemple Dans l'implantation d'un arbre couvrant dans laquelle chacun des processeurs possède comme variable la distance à la racine, le détecteur de défaillance peut utiliser la variable de distance pour détecter des configurations incorrectes : si chaque processeur possède une distance à la racine qui est plus grande de un que la distance à la racine de son père et que la racine n'a pas de père, alors le système est dans un état correct. Au lieu de nécessiter une vue à distance $\lceil \frac{n}{4} \rceil$, une simple consultation du nœud parent suffit, soit une vue à distance 1.

Une technique de surveillance qui peut être utilisée comme un détecteur de défaillances pour des algorithmes est présentée dans [AD97]. Cette technique permet de détecter les inconsistances dans chaque algorithme, qu'il soit ou non interactif. Etant donné que cette technique est universelle, il est possible de concevoir des détecteurs de défaillances qui soient plus efficaces (en terme de mémoire) pour un ensemble spécifique d'algorithmes.

Hiérarchie des Détecteurs pour les Algorithmes Une hiérarchie pour les détecteurs de défaillances d'algorithmes peut être définie de manière analogue à celle que nous avons définie pour les détecteurs de défaillances pour des tâches. Le choix de l'algorithme qui implante une tâche est déterminant lors du choix d'un détecteur de défaillances adéquat.

Exemple Il est possible de suggérer l'utilisation d'un algorithme de mise à jour de topologie pour implanter les tâches statiques mentionnées plus haut. Un algorithme de mise à jour de topologie permet à chaque processeur de connaître la topologie du système tout entier, de telle sorte que chacun des processeur est capable d'exécuter des calculs *locaux* afin d'élire un chef ou de compter le nombre de nœuds présents dans le système. Clairement, ce parti-pris dans le choix de l'implantation consomme plus de mémoire que toutes les autre implantations possibles. D'un autre coté, il est possible de détecter des incohérences éventuelles en utilisant un détecteur de défaillances de l'ensemble $\mathcal{FD}_1$.

3.7 Implantation

Dans cette section, nous donnons une implantation possible de l'utilisation et de la mise à jour des variables locales des détecteurs de défaillances transitoires, à savoir la *vue locale* (pour les algorithmes statiques) et l'*histoire locale* (pour les algorithmes dynamiques). Nous supposons ici que chaque processeur P_i exécute simultanément l'algorithme résolvant la tâche et le détecteur de défaillances transitoires pour cette tâche.

Mise à Jour de la Vue Chaque processeur P_i communique continuellement à chacun de ses voisins P_j la portion de $\mathcal{V}_i^d$ qui est partagée avec $\mathcal{V}_j^d$. En d'autres termes, P_i n'envoie pas à P_j ses informations à propos des composants du système qui sont à distance $d+1$ de P_j (en accord avec la portion du graphe de communication stocké dans $\mathcal{V}_i^d$). Lorsque P_i reçoit l'information $\mathcal{V}_j^d$ de son voisin P_j, P_i vérifie que les portions partagées par $\mathcal{V}_i^d$ et $\mathcal{V}_j^d$ sont identiques. P_i émet un avis d'erreur si ce n'est pas le cas.

Il est facile de constater que le test ci-dessus assure que chacun des processeurs du système possède une vue correcte des composants du système à distance d. Supposons que ce ne soit pas le cas de la vue du processeur P_i vis à vis du processeur P_k. Soient $P_i, P_{j1}, P_{j2}, \cdots, P_k$ les processeurs apparaîssant sur un plus court chemin (de longueur inférieure ou égale à d) de P_i à P_k. Soit P_{jl} le premier processeur sur ce chemin tel que les variables $\mathcal{V}_{jl}$ et $\mathcal{V}_i$ possèdent des valeurs différentes à propos d'une variable de P_k. Il existe forcément un tel processeur puisque P_i et P_k possèdent des valeurs différentes pour les variables de P_k. Alors, P_{jl} et $P_{j(l-1)}$ peuvent émettre un avis d'erreur.

Mise à Jour de l'Histoire La politique de mise à jour du tableau d'histoire locale est la suivante : à chaque étape synchrone, la dernière vue $\mathcal{V}_i^d[s]$ devient la première vue, et chacune des autres vues $\mathcal{V}_i^d[j]$ est copiée dans $\mathcal{V}_i^d[j+1]$, pour $1 \leq j < s$.

Le dernier point pertinent lors de l'implantation d'un détecteur de défaillances transitoires est l'action à effectuer lors de l'émission d'un avis d'erreur. Bien que ce soit hors du cadre de cette thèse, nous mentionnons la réinitialisation (par exemple [KP93] ou [APSVD94]) et la réparation (par exemple [AD97] et [KPS97]), qui devraient toutes deux rétablir le système dans une configuration correcte, ainsi que le détecteur de défaillances associé. Bien entendu, le détecteur de défaillances ne doit pas être réactivé avant que la réinitialisation ou la réparation ait eu lieu.

3.8 Résumé

Dans ce chapitre, nous avons déterminé quelles étaient les ressources nécessaires à l'implantation de détecteurs de défaillances transitoires pour des tâches ou des algorithmes. Nous avons présenté une hiérarchie des détecteurs de défaillances transitoires qui détectent l'occurence d'une ou plusieurs défaillances en *un seul* cycle de calcul (dans un système asynchrone) ou en une seule étape de calcul (dans un système synchrone). La hiérarchie présentée identifie la *localité* des tâches et des algorithmes comme étant la quantité d'information à transmettre pour construire un détecteur de défaillances transitoires.

Il faut toutefois remarquer que dans l'implantation proposée, un détecteur de défaillances transitoire n'est pas fiable puisqu'il peut détecter une erreur quand les fautes corrompent l'état du détecteur lui même et non l'état du système. La seule garantie que nous puissions avoir, c'est que *(i)* si le système se trouve dans un état incohérent, une erreur sera détectée, et *(ii)* si à la fois le système et le détecteur se trouvent dans un état correct, aucune erreur ne sera détectée.

Chapitre 4

Auto-stabilisation sans Surcoût

Au chapitre 3, nous avons déterminé une borne supérieure à la mémoire devant être utilisée par un détecteur de défaillances transitoires pour de nombreuses tâches statiques. Ce détecteur peut être greffé sur un algorithme réparti classique pour le rendre auto-stabilisant, mais il induit un surcoût mémoire qui, la plupart du temps, est loin d'être négligeable. Une deuxième approche, celle qui est l'objet de ce chapitre, consiste à ne pas modifier les algorithmes répartis classiques, et à trouver des conditions sous lesquelles ils sont naturellement auto-stabilisants.

Plus précisément, nous donnons une condition locale garantissant qu'un système réparti est auto-stabilisant. Nos résultats positifs portent uniquement sur les algorithmes résolvant des spécifications *statiques*. Comme notre solution est basée sur une condition locale, il n'existe pas de niveau algorithmique supplémentaire utilisé pour rendre un algorithme auto-stabilisant. De fait, lorsqu'aucune défaillance transitoire n'apparaît dans le système, les performances de l'algorithme ne souffrent d'aucun surcoût.

Dans ce chapitre, nous résolvons plusieurs tâches statiques de manière auto-stabilisante dans un système dont le graphe de communication est véritablement quelconque. En particulier, aucune hypothèse n'est faite sur la forte connexité ou la présence de cycles. A la différence de nombreuses approches, notre solution ne requiert aucune connaissance sur le graphe de communication, comme sa taille, son diamètre ou encore son degré maximum.

4.1 Références

Historiquement, la recherche en auto-stabilisation a sutout concerné les systèmes dont le graphe de communication est non-orienté et où des communications bidirectionelles sont possibles (le protocole *Update* de [DH97], ou bien les algorithmes présentés dans [DIM93, AKY90]). Les communication bidirectionnelles trouvent leur usage dans les systèmes répartis auto-stabilisants pour comparer l'état d'un nœud avec celui de ses voisins et vérifier la consistance du système. Les algorithmes auto-stabilisants qui sont construits à partir du paradigme de la détection locale (par exemple [APSV91, APSVD94]) utilisent ces facilités.

Le défaut de bidirectionnalité des communications a été surmonté dans plusieurs travaux récents et utilisant plusieurs techniques. La forte connexité (qui est un prérequis plus faible que la bidirectionnalité) est supposée pour permettre de construire une topologie bien connue sur laquelle un algorithme auto-stabilisant peut être construit facilement. Comme de nombreux algorithmes auto-stabilisants existent pour des réseaux en anneau ([Dij74]) ou en arbre ([ABDT98]) dans la litterature, ces constructions peuvent être utilisées pour réutiliser des algorithmes existants dans des réseaux généraux.

Se restreindre à des graphes de communication bidirectionnels ou fortement connexes est raisonnable quand la tâche à résoudre est dynamique et que le système est asynchrone : par exemple, pour des algorithmes de passage de jeton, un message doit être en mesure de passer par chacun des processeurs du système un nombre infini de fois. Cependant, il existe plusieurs tâches statiques pour lesquelles des communications globales ne sont pas indispensables. Par exemple, le calcul de l'arbre des plus courts chemins à partir d'une source unique requiert simplement qu'un chemin existe à partir d'un nœud particulier vers chacun des autres nœuds, mais pas l'inverse.

Dans [AAEG93], Arora *et al.* utilisent le formalisme des systèmes d'itérations pour donner des conditions suffisantes pour la convergence des systèmes répartis auto-stabilisants. La modélisation adoptée ne considère que les exécutions équitables du système sous le démon réparti. Les conditions proposées sont indépendantes de la tâche à effectuer, mais elles sont globales (elles portent sur le système tout entier) et dynamiques (elles portent sur les exécutions du système). Avoir seulement une condition globale dynamique implique que pour garantir qu'un système particulier est auto-stabilisant, il est nécessaire de mettre en place une plate-forme de test, d'exécuter le système autant de fois que nécessaire pour constater que ses exécutions satisfont la propriété attendue. Cette approche est, on le voit, extrèmement contraignante, et n'est valable que pour un système particulier. Si on change la topologie du système alors toute la procédure de test doit être effectuée de nouveau, avant de pouvoir conclure à des propriétés de stabilisation.

Dans [Her91], Herman présente deux algorithmes (un pour le calcul d'un arbre de poids minimal, et un pour le calcul d'un arbre en profondeur) qui sont généralisés dans [Tel94] aux systèmes répartis dont les exécutions sont équitables et s'exécutent sous le démon réparti en donnant des conditions globales sur le système réparti. A la différence des conditions exprimées dans [AAEG93], les conditions de Tel sont spécifiques à un système particulier, mais elles peuvent être vérifiées statiquement (elles s'appliquent sur les configurations et les transitions du système). Avoir une ou plusieurs conditions locales statiques sur un système permet de prouver qu'un système particulier est stabilisant sans avoir besoin de mettre en place une plate forme de test où le système s'exécute. Toutefois, si le système est modifié même légèrement (par exemple par un changement de topologie), toutes les preuves de vérification de conditions doivent être effectuées à nouveau.

Notre solution est indépendante de la tâche à résoudre, et dans une moindre mesure du système, puisqu'elle est entièrement déterminée par les propriétés algébriques de l'algorithme

calculé localement sur chaque processeur. Autrement dit, nous nous basons uniquement sur une condition statique locale. Nous proposons un algorithme paramétré qui peut être instancié par une fonction locale. Cet algorithme permet à un ensemble de tâches statiques de trouver une solution auto-stabilisante pourvu que l'algorithme local puisse s'exprimer au moyen de primitives locales de calcul appelées des r-opérateurs. Les r-opérateurs sont suffisamment généraux pour que des applications comme le calcul des plus courts chemins à partir d'une source ou la construction d'un arbre en profondeur puisse être résolue par des systèmes répartis auto-stabilisants dont le graphe de communication est quelconque. Avoir une condition locale statique sur un algorithme permet de prouver que tous les systèmes dont les processeurs ont un code qui répond à cette condition sont auto-stabilisants, indépendament de la topologie. La preuve n'a pas à être modifiée pour des configurations particulières du système.

4.2 Hypothèses Spécifiques au Chapitre

Dans ce chapitre, nous présentons un algorithme paramétré par une fonction locale. Dans le cas où le graphe de communication du système est orienté et sans cycle, la fonction peut être quelconque, et dans le cas général, la fonction doit être un r-opérateur vérifiant certaines propriétés. Dans cette section, nous donnons quelques notations concernant les graphes orientés acycliques (section 4.2.1) et nous présentons brièvement les r-opérateurs et leurs propriétés (section 4.2.2).

4.2.1 Graphes Orientés Acycliques

Considérer un graphe sans cycle peut présenter un intéret dans le cas où le graphe de communication est non-orienté et que l'on y construit une orientation acyclique. Si on dispose d'indentifiants uniques sur chacun des processeurs du système, alors il est possible de construire une orientation acyclique comme suit en définissant un ordre total $<$ sur les identifiants (`id`) : un processeur P_i compare son $\texttt{id}_i$ avec celui de son voisin P_j. Si $\texttt{id}_i < \texttt{id}_j$, alors P_i suppose que P_j est l'un de ses prédécesseurs immédiats. Dans le cas contraire, P_j est un successeur immédiat. Puisque $<$ définit un ordre total, les relations $\texttt{id}_i < \texttt{id}_j < \texttt{id}_k$ et $\texttt{id}_k < \texttt{id}_i$ ne peuvent exister entre trois identifiants quelconque. Par conséquent, aucun cycle n'est possible dans le graphe orienté résultant de cette oprétation. Ghosh et Karaata (dans [GK93]) ont présenté un protocole auto-stabilisant d'orientation acyclique pour les graphes planaires (dont le degré est au maximum 6).
L'ensemble des chemins de i à j est dénoté par $X_{i,j}$. Tous les ascendants de i (lui-même compris) sont dits *atteignables* de i. Dans un graphe orienté acyclique, il est utile de définir la notion de niveau à partir des sources du graphe au moyen d'une méthode récursive :

Définition 4.1 (Niveau) *Le* niveau l_i *d'un nœud* i *d'un graphe orienté acyclique* $\mathcal{G}$ *est*

défini par :

$$l_i \equiv \begin{cases} 0 & si\ \Gamma_{\mathcal{G}}^{-1}(i) = \emptyset \\ 1 + \mathsf{Max}_{j \in \Gamma_{\mathcal{G}}^{-1}(i)}\{l_j\} & sinon \end{cases}$$

Les nœuds source (qui n'ont pas d'ascendants) sont au niveau 0, tandis que les autres nœuds sont de niveau égal au plus haut niveau de chacun de leurs ascendants directs plus un. Par exemple, un nœud i dont les ascendants direct sont uniquement des sources est au niveau $l_i = 1$, et un nœud k dont les ascendants directs sont respectivement aux niveaux 1, 5, et 8 est au niveau $l_k = 9$.

4.2.2 r-Opérateurs

Dans cette section, nous définissons les r-opérateurs comme une extension des Infima de Tel. Les r-opérateurs sont définis comme des fonctions sur un ensemble de valeurs $\mathbb{S}$ qui possèdent plusieurs propriétés algébriques. Ces r-opérateurs sont utilisés au cœur de notre algorithme afin de le rendre automatiquement auto-stabilisant.

Les Infima

Dans ce chapitre, nous considérons un algorithme unique paramétré qui exécute un opérateur sur des données reçues en entrée. De tels opérateurs sont ensuite suffisants pour décrire le comportement de tout le système. Dans [Tel94], Tel a montré que de tels calculs terminaient lorsque l'algorithme était instancié par un Infimum (ici appelé s-opérateur) sur l'ensemble des entrées.

Définition 4.2 (s-opérateur) *L'opérateur $\oplus$ est un* s-opérateur *sur l'ensemble $\mathbb{S}$ si $\oplus$ est un opérateur binaire associatif, commutatif et idempotent.*

Exemple L'opérateur min retournant le minimum de deux entiers naturels est un s-opérateur sur l'ensemble $\mathbb{N}$.

Un tel opérateur définit une relation d'ordre partiel $\preceq_{\oplus}$sur l'ensemble $\mathbb{S}$ de la manière suivante : $x \preceq_{\oplus} y$ si et seulement si $x \oplus y = x$. De plus, [Tel94] suppose qu'il existe un plus grand élément sur $\mathbb{S}$, dénoté par $e_{\oplus}$, qui vérifie $x \preceq_{\oplus} e_{\oplus}$ pour tout $x \in \mathbb{S}$. Si nécessaire, cet élément peut être ajouté à $\mathbb{S}$. Dans ce qui suit, nous supposons toujours qu'un s-opérateur admet un tel plus grand élément dans son ensemble de définition $\mathbb{S}$.

Utiliser un s-opérateur $\oplus$ comme paramètre de notre algorithme paramétré induit un système silencieux. Néanmoins, des contre-exemples montrent que ce système n'est pas auto-stabilisant (voir section 4.4.3).

Les r-Opérateurs

Dans [Duc99], Ducourthial propose une algèbre déformée — la r-algèbre — qui généralise les résultats de Tel. Un r-opérateur introduit une fonction appliquée au deuxième argument en préalable à l'évaluation du s-opérateur.

Définition 4.3 (r-opérateur) *L'opérateur $\triangleleft$ est un r-opérateur sur l'ensemble $\mathbb{S}$ s'il existe un s-opérateur $\oplus$ sur $\mathbb{S}$ et un homomorphisme r de $(\mathbb{S}, \oplus)$ — appelé une r-fonction — tels que $\triangleleft$ vérifie, $\forall x, y \in \mathbb{S}$, $x \triangleleft y = x \oplus \mathsf{r}(y)$.*

Plusieurs propriétés des r-opérateurs rendent compte de la dissymétrie introduite par la r-fonction dans le s-opérateur sur lequel il est basé.

Proposition 4.1 *Soit $\triangleleft$ un r-opérateur sur $\mathbb{S}$, basé sur le s-opérateur $\oplus$ et la r-fonction r. Alors $\triangleleft$ vérifie les propriétés suivantes : **(i)** r-associativité : $(x \triangleleft y) \triangleleft \mathsf{r}(z) = x \triangleleft (y \triangleleft z)$; **(ii)** r-commutativité : $\mathsf{r}(x) \triangleleft y = \mathsf{r}(y) \triangleleft x$; **(iii)** r-idempotence : $\mathsf{r}(x) \triangleleft x = \mathsf{r}(x)$.*

Preuve: (i) $\forall x, y, z \in \mathbb{S}$, $(x \triangleleft y) \triangleleft \mathsf{r}(z) = x \oplus \mathsf{r}(y) \oplus \mathsf{r}^2(z) = x \oplus \mathsf{r}(y \oplus \mathsf{r}(z)) = x \triangleleft (y \triangleleft z)$.
(ii) $\forall x, y \in \mathbb{S}$, $\mathsf{r}(x) \triangleleft y = \mathsf{r}(x) \oplus \mathsf{r}(y) = \mathsf{r}(y) \oplus \mathsf{r}(x) = \mathsf{r}(y) \triangleleft x$
(iii) $\forall x \in \mathbb{S}$, $\mathsf{r}(x) \triangleleft x = \mathsf{r}(x) \oplus \mathsf{r}(x) = \mathsf{r}(x)$. □

La réciproque peut être montrée lorsque la r-fonction est bijective. Par conséquent, un r-opérateur est un s-opérateur dissymétrique dans le cas général, et un s-opérateur quand la r-fonction est égale à l'identité (c'est à dire $\forall x \in \mathbb{S}, \mathsf{r}(x) = x$). Nous rappelons maintenant certaines propriétés importantes des r-opérateurs.

Proposition 4.2 *La r-fonction r d'un r-opérateur $\triangleleft$ est un homomorphisme de $(\mathbb{S}, \triangleleft)$.*

Preuve: $\forall x, y \in \mathbb{S}$, $\mathsf{r}(x \triangleleft y) = \mathsf{r}(x \oplus \mathsf{r}(y)) = \mathsf{r}(x) \oplus \mathsf{r}^2(y) = \mathsf{r}(x) \triangleleft \mathsf{r}(y)$. □

Proposition 4.3 *Soit $\triangleleft$ un r-opérateur sur $\mathbb{S}$ basé sur le s-opérateur $\oplus$ et la r-fonction r. Alors la r-fonction r est déterminée de manière unique par $\mathsf{r}(x) = e_\oplus \triangleleft x$.*

Preuve: $\mathsf{r}(x) = e_\oplus \oplus \mathsf{r}(x) = e_\oplus \triangleleft x$. □

Proposition 4.4 *Le r-opérateur $\triangleleft$ basé sur le s-opérateur $\oplus$ est idempotent si et seulement si $\forall x \in \mathbb{S}$, $x \preceq_\oplus \mathsf{r}(x)$.*

Preuve: $x \triangleleft x = x \Leftrightarrow x \oplus \mathsf{r}(x) = x \Leftrightarrow x \preceq_\oplus \mathsf{r}(x)$ □

De la proposition précédente, nous introduisons le terme d'*idempotence stricte*, utilisé dans la suite du chapitre. En effet, si l'opérateur qui caractérise notre système est un r-opérateur, alors le système est auto-stabilisant si et seulement si ce r-opérateur est strictement idempotent.

Définition 4.4 (Stricte Idempotence) *Le r-opérateur $\triangleleft$ basé sur le s-opérateur $\oplus$ est* strictement idempotent *si et seulement si $\forall x \in \mathbb{S} \setminus \{e_\oplus\}$, $x \prec_\oplus \mathsf{r}(x)$ (c'est à dire $x \preceq_\oplus \mathsf{r}(x)$ et $\mathsf{r}(x) \neq x$).*

Notons que par définition, tout r-opérateur strictement idempotent est également idempotent.

Proposition 4.5 *Soit $\triangleleft$ un r-opérateur idempotent sur $\mathbb{S}$ basé sur le s-opérateur $\oplus$ et la r-fonction r. Alors $\mathsf{r}(e_\oplus) = e_\oplus$.*

Preuve: Puisque $\triangleleft$ est idempotent, $e_\oplus \oplus \mathsf{r}(e_\oplus) = e_\oplus$. Puisque $e_\oplus$ est l'élément neutre de $\oplus$, $e_\oplus \oplus \mathsf{r}(e_\oplus) = \mathsf{r}(e_\oplus)$. □

Proposition 4.6 *L'élément neutre $e_\oplus$ d'un s-opérateur $\oplus$ est l'élément neutre à droite (c'est à dire $x \triangleleft e_\oplus = x$) de tous les r-opérateurs idempotents basés sur $\oplus$.*

Preuve: $\forall x \in \mathbb{S}$, $x \triangleleft e_\oplus = x \oplus \mathsf{r}(e_\oplus) = x \oplus e_\oplus = x$. □

Exemple L'opérateur $\mathsf{minc}(x, y) \equiv \mathsf{min}(x, y + 1)$ est un r-opérateur strictement idempotent sur $\mathbb{Z} \cup \{+\infty\}$, en considérant $+\infty$ comme son élément neutre. Il est basé sur le s-opérateur min et sur la r-fonction bijective $\mathsf{r}(x) \equiv x + 1$. Un tel opérateur peut également être défini sur un ensemble fini tel $\{0, 1, \ldots, 255\}$. Dans ce cas, la r-fonction n'est pas bijective, mais définie par $\mathsf{r}(x) \equiv x + 1$ pour $x \in \{0, \ldots, 254\}$ et $\mathsf{r}(255) \equiv 255$. Même dans le cas fini, le r-opérateur reste strictement idempotent.

Les r-opérateurs binaires peuvent être étendus pour accepter un nombre arbitraire d'arguments :

Définition 4.5 (r-opérateur n-aire) *L'opérateur $\triangleleft$ est un r-opérateur n-aire de $\mathbb{S}^n$ dans $\mathbb{S}$ si et seulement s'il existe un s-opérateur $\oplus$ sur $\mathbb{S}$ et $n-1$ homomorphismes $\mathsf{r}_1, \ldots, \mathsf{r}_{n-1}$ de $(\mathbb{S}, \oplus)$ tels que $\triangleleft$ vérifie $\forall x_0, \ldots, x_{n-1} \in \mathbb{S}$, $\triangleleft(x_0, \ldots, x_{n-1}) = x_0 \oplus \mathsf{r}_1(x_1) \oplus \cdots \oplus \mathsf{r}_{n-1}(x_{n-1})$.*

En d'autres termes, un r-opérateur n-aire consiste en $n-1$ r-opérateurs binaires basés sur le même s-opérateur. Si chacun de ces r-opérateurs binaires est (strictement) idempotent, le r-opérateur n-aire résultant est (strictement) idempotent :

Définition 4.6 *Le r-opérateur n-aire $\triangleleft$ de $\mathbb{S}^n$ dans $\mathbb{S}$ basé sur le s-opérateur $\oplus$ et les $n-1$ homomorphismes $\mathsf{r}_1, \ldots, \mathsf{r}_{n-1}$ de $(\mathbb{S}, \oplus)$ est* idempotent *(respectivement* strictement idempotent*) si pour tout $x \in \mathbb{S}$ (respectivement $x \in \mathbb{S} \setminus \{e_\oplus\}$), et pour toute r-fonction r_i ($1 \leq i < n$), $x \preceq_\oplus \mathsf{r}_i(x)$ (respectivement $x \prec_\oplus \mathsf{r}_i(x)$, c'est à dire $x \preceq_\oplus \mathsf{r}_i(x)$ et $\mathsf{r}_i(x) \neq x$).*

Un r-opérateur n-aire dont toutes les r-fonctions sont identiques peut être vu comme un r-opérateur binaire étendu à n opérandes.

Exemple L'expression $\mathsf{min}(x_0, x_1 + 1, \ldots, x_{n-1} + 1)$ définit un r-opérateur n-aire, mais peut également s'écrire en utilisant le r-opérateur minc comme suit :

$$\mathsf{minc}(\mathsf{minc}(\cdots(\mathsf{minc}(x_0, x_1), \ldots x_{n-3}), x_{n-2}), x_{n-1})$$

Ce qui se réécrit en : $x_0 \triangleleft x_1 \triangleleft \cdots \triangleleft x_{n-1}$ où $x \triangleleft y$ dénote $\mathsf{minc}(x, y)$.

Quand chacun des nœuds exécute des calculs locaux en utilisant un r-opérateur, qu'il soit binaire ou n-aire, chaque arc du graphe de communication du système réparti est associé à une r-fonction (voir figure 4.1). Quand les calculs locaux sont exécutés au moyen d'un s-opérateur, les r-fonctions sont égales à l'identité puisque les données entrantes ne sont pas modifiées. L'expression *variable r-augmentée* dénote le résultat d'une r-fonction associée à un arc entrant. Dans la suite, et lorsque c'est possible, nous utilisons un r-opérateur binaire étendu au nombre approprié d'opérandes, comme ci dessus avec l'opérateur minc. Ce nombre d'opérandes est égal à un de plus que le nombre d'arcs incidents du nœud considéré (c'est à dire $\delta^-(v) + 1$ pour le nœud v).

Dans ce chapitre, nous supposons que les hypothèses suivantes sont vérifiées concernant les opérateurs utilisés :

Hypothèse 4.1 (Ordre Total) *Le s-opérateur $\oplus$ induit un ordre* total *sur $\mathbb{S}$.*

Hypothèse 4.2 (Ensemble Fini) *L'ensemble $\mathbb{S}$ de définition des opérateurs est fini.*

La section 4.6 propose un relâchement de l'hypothèse 4.2 pour traiter le cas des ensembles infinis. Notons que toutes les applications présentées dans ce chapitre restent valables avec un tel relâchement.

4.3 Algorithme Paramétré

Nous présentons dans cette section un algorithme paramétré par une fonction locale à chaque processeur. Après avoir informellement décrit le fonctionnement de l'algorithme dans la section 4.3.1, nous donnons les détails d'implantation dans la section 4.3.2.

4.3.1 Description Informelle

Chaque processeur P_v possède deux constantes locales stockées en ROM : la *donnée initiale*, $M_v[0]$, et l'ensemble de ses ascendants directs $\Gamma_{\mathcal{G}}^{-1}(v)$. Pour répliquer les valeurs associées à ses ascendants directs et garantir le bon fonctionnement de l'algorithme sous un démon à lecture-écriture, P_v possède également $\delta^-(v)$ variables locales $M_v[1]$, ..., $M_v[\delta^-(v)]$, appelées *variables d'entrée*. Pour stocker le résultat de chaque calcul local, un unique registre est utilisé par P_v : $M_v[\delta^-(v) + 1]$, la *variable de sortie*.

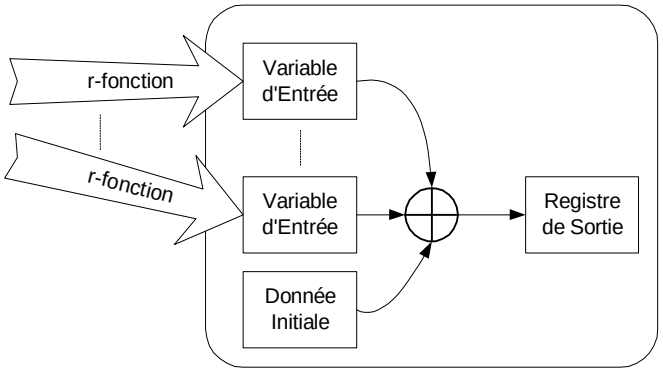

FIG. 4.1 – Description d'un processeur P_v

En plus de ce qui précède, chaque processeur P_v dispose d'une fonction locale $\mathcal{F}_v$ définie comme suit :

$$\begin{array}{rccl} \mathcal{F}_v : & \mathbb{S}^{\delta^-(v)+1} & \rightarrow & \mathbb{S} \\ & M_v[0..\delta^-(v)] & \mapsto & \mathcal{F}(M_v[0..\delta^-(v)]) \end{array}$$

La fonction $\mathcal{F}_v$ implante le calcul local d'un opérateur $(\delta^-(v)+1)$-aire. Les entrées de $\mathcal{F}_v$ sont la donnée initiale $M_v[0]$ ainsi que les valeurs des copies les plus récentes des variables de sortie des ascendants directs, stockées dans $M_v[1..\delta^-(v)]$. Le résultat de la fonction $\mathcal{F}$ constitue l'état courant du processeur exécutant la fonction ; il est stocké dans la variable de sortie $M_v[\delta^-(v)+1]$.

Localement, le code exécuté par chaque processeur P_v effectue les actions suivantes :

(1) lecture des données dans les registres des ascendants directs et copie de ces données dans les variables d'entrée $M_v[1], M_v[2], \ldots M_v[\delta^-(v)]$.

(2) calcul local en utilisant la fonction $\mathcal{F}_v$, la donnée initiale $M_v[0]$ et les variables d'entrée $M_v[1], \ldots M_v[\delta^-(v)]$, puis stockage du résultat du calcul dans la variable de sortie $M_v[\delta^-(v)+1]$:

$$M_v[\delta^-(v)+1] \quad \leftarrow \quad \mathcal{F}_v\left(M_v[0], \ldots, M_v[\delta^-(v)]\right)$$

La figure 4.1 représente le processeur P_v avec ses registres M_v et les r-fonctions qui sont associées à chacun des arcs entrants de v.

4.3.2 Algorithme

Plus formellement, l'algorithme paramétré que nous proposons consiste en deux règles gardées instanciées par $\mathcal{F}_v$ comme décrit dans l'algorithme 1.

Algorithme 1 *L'algorithme de chaque processeur P_v est le suivant :*

Copie
$R_1|\mathcal{F}_v$: $\left(P_u \in \Gamma_{\mathcal{G}}^{-1}(v) : \left(\alpha := \text{Lit}\left(M_u[\delta^-(u)+1]\right)\right) \wedge M_v[Ind(u)] \neq \alpha\right) \longrightarrow$
$M_v[Ind(u)] := \alpha;$

Calcul
$R_2|\mathcal{F}_v$: $\left(\text{Lit}\left(M_v[\delta^-(v)+1]\right) \neq \mathcal{F}_v\left(M_v[0..\delta^-(v)]\right)\right) \longrightarrow$
$\text{Ecrit}\left(M_v[\delta^-(v)+1], \mathcal{F}_v\left(M_v[0..\delta^-(v)]\right)\right);$

La première règle, appelée *règle de copie*, est dénotée par $R_1|\mathcal{F}_v$. Elle duplique le contenu de la variable de sortie d'un ascendant direct dans une variable locale, de manière à pouvoir s'en servir plus tard. Elle utilise une variable locale α de manière à éviter la lecture du registre partagé $M_u[\delta^-(v)+1]$ deux fois (ce qui peut poser des problèmes si le système fonctionne sous un démon à lecture-écriture). Dans la règle $R_1|\mathcal{F}_v$, une expression de la forme $\langle gauche\rangle := \langle droit\rangle$ est utilisée. Le premier opérande doit être une variable et le deuxième peut être une constante ou une variable. L'opérateur := assigne la valeur placée dans *droit* à la variable *gauche* et retourne toujours *vrai*.

La seconde règle, appelée *règle de calcul* et dénotée par $R_2|\mathcal{F}_v$, calcule la fonction $\mathcal{F}_v$ au moyen des variables précédemment copiées comme opérandes et stocke le résultat de l'appel de la fonction dans $M_v[\delta^-(v)+1]$.

Il est notable que chacune de ces deux règles est *auto-inhibitrice*. Plus précisément, aucune règle ne peut être exécutée deux fois de suite par un même processeur sans un changement interne ou externe. Ceci est utilisé pour garantir que tout journal d'exécution du système est silencieux même sous un démon adéquat (c'est à dire non-équitable).

Notes d'implantation

Disposer de l'ensemble $\Gamma_{\mathcal{G}}^{-1}(v)$ sur chaque processeur P_v du système est utile seulement pour la description technique de l'algorithme. Dans une implantation effective, cet ensemble peut être efficacement remplacé par la liste des arcs entrants de v.

4.4 Preuve de Correction

Nous souhaitons montrer que l'algorithme 1 fonctionne correctement indépendamment du démon contraignant les exécutions (nous supposons que le système s'exécute sous un démon lecture-écriture), de l'équité des exécutions (nous considérons des exécutions simplement maximales), de la topologie du graphe de communication du système (nous traitons le cas des graphes orientés acycliques dans la section 4.4.2 et le cas général dans la section 4.4.3).

4.4.1 Atomicité de Lecture/Ecriture

Dans la suite du chapitre, nous supposons que les règles de l'algorithme 1 sont exécutées atomiquement. Or, les règles de l'algorithme ont respectivement une action *Lit* (pour la règle

$R_1|\mathcal{F}_v$) et une action *Lit* et une action *Ecrit* (pour la règle $R_2|\mathcal{F}_v$). Il faut donc montrer que ces deux règles, même considérées comme atomiques, n'entravent pas le bon fonctionnement de l'algorithme sous le démon lecture-écriture tel que décrit dans [DIM93].

Lemme 4.1 *Les règles gardées $R_1|\mathcal{F}$ et $R_2|\mathcal{F}$ du système $\mathcal{PA}$ peuvent être considérées comme atomiques.*

Preuve: La règle $R_1|\mathcal{F}$ contient seulement une action *Lit* et des actions internes, et par conséquent peut être considérée comme atomique sous le démon lecture-écriture.

La règle $R_2|\mathcal{F}$ contient une action *Lit* et une action *Ecrit*, toutes deux utilisant le même registre reg_u. Sous un démon à lecture-écriture, ces deux actions peuvent être entrelacées avec d'autres actions d'autres processeurs. Prouvons maintenant que chacun des entrelacement possibles est équivalent à une exécution où la règle $R_2|\mathcal{F}$ est exécutée atomiquement.

Soit P_u le processeur propriétaire du registre reg_u et soit $\Gamma_{\mathcal{G}}^{+1}(u)$ l'ensemble de ses descendants directs. L'entrelacement de la règle $R_2|\mathcal{F}$ sur le processeur P_u et de la règle $R_2|\mathcal{F}$ sur le processeur P_v (avec $v \neq u$) est équivalent à l'exécution atomique de ces deux règles dans un ordre quelconque, puisqu'elles s'appliquent à des registres différents ($R_2|\mathcal{F}$ sur le processeur P_u applique chacune de ses actions sur le même registre reg_u). Pour des raisons analogues, l'entrelacement des règles $R_2|\mathcal{F}$ sur le processeur P_u et $R_1|\mathcal{F}$ sur le processeur P_v ($v \notin \Gamma_{\mathcal{G}}^{+1}(u)$) est équivalent à leur exécution atomique dans un ordre quelconque.

Reste le cas où la règle $R_2|\mathcal{F}$ sur le processeur P_u est entrelacée avec la règle $R_1|\mathcal{F}$ sur le processeur P_v (avec $v \in \Gamma_{\mathcal{G}}^{+1}(u)$). Les deux actions peuvent impliquer le même registre reg_u, et la trace résultante serait de la forme :

$\ldots, Lit_u(\text{reg}_u), Lit_v(\text{reg}_u), Ecrit_u(\text{reg}_u), \ldots$

Cette trace est équivalente à la suivante où la règle $R_2|\mathcal{F}$ sur le processeur P_u (qui inclut les actions $Lit_u(\text{reg}_u)$ et $Ecrit_u(\text{reg}_u)$) est exécutée atomiquement :

$\ldots, Lit_v(\text{reg}_u), Lit_u(\text{reg}_u), Ecrit_u(\text{reg}_u), \ldots$

Dans tous les cas, chacune des traces entrelacées est équivalente à une trace où la règle $R_2|\mathcal{F}$ est exécutée de manière atomique. □

4.4.2 Cas des Graphes sans Cycles

Dans [AAEG93], les auteurs ont montré qu'un algorithme réparti construit à partir d'une fonction locale était naturellement auto-stabilisant pour peu que l'on considère uniquement les exécutions équitables sous le démon réparti du système. Dans cette section, nous étendons ce résultat en montrant que les exécutions adéquates sous le démon lecture-écriture sont également admissible pour peu que la fonction locale soit conditionnée par l'algorithme 1.

Soit N_k l'ensemble des nœuds dont le niveau est k. Une configuration du système est *DAG*-légitime si elle appartient à l'ensemble $\mathcal{L}_{\mathcal{F}}$ défini comme :

$$\mathcal{L}_{\mathcal{F}} \equiv \left\{ \begin{array}{l} \forall i \in N_0, M_i[\delta^-(i)+1] = M_i[0] \\ \forall j \geq 1, \forall i \in N_j, M_i[\delta^-(i)+1] = \mathcal{F}(M_i[0..\delta^-(i)]) \\ \forall j \geq 1, \forall i \in N_j, \forall k \in \Gamma_{\mathcal{G}}^{+1}(i), M_i[Ind(k)] = M_k[\delta^-(k)+1] \end{array} \right\}$$

Nous prouvons les deux lemmes suivants :

Lemme 4.2 (Clôture) *En partant d'une configuration de l'ensemble $\mathcal{L}_{\mathcal{F}}$ et en supposant que $\mathcal{F}$ soit déterministe, toute exécution du système conserve $\mathcal{L}_{\mathcal{F}}$.*

Preuve: Puisqu'aucune règle n'est applicable, la configuration de $\mathcal{L}_{\mathcal{F}}$ reste la même. □

Lemme 4.3 *Dans toute exécution de $\mathcal{S}$, chaque processeur exécute un nombre fini d'actions.*

Preuve: La preuve se fait par récurrence sur le niveau du processeur.

Si le processeur P_i est de niveau 0, alors seule son action $R_2|\mathcal{F}$ est éventuellement exécutable (si sa variable de sortie est différente de sa valeur finale suite à une défaillance transitoire). Une fois que l'action $R_2|\mathcal{F}$ a été exécutée, aucune action n'est possible sur le processeur P_i.

Supposons qu'il existe un entier n supérieur ou égal à 0 tel que tous les processeurs de niveau inférieur ou égal à n exécutent un nombre fini d'actions, et montrons que tous les processeurs de niveau $n+1$ exécutent eux aussi un nombre fini d'actions. Considérons un processeur P_j de niveau $n+1$ et plaçons-nous dans une configuration C_n où tous les processeurs de niveau inférieur ou égal à n n'exécutent plus aucune action (par hypothèse de récurrence, une telle configuration C_n existe toujours). Le processeur P_j possède $\delta^-(j)$ ascendants directs dont la variable de sortie ne correspond pas forcément à la vue que P_j en a. Donc la règle $R_1|\mathcal{F}$ est potentiellement exécutable $\delta^-(j)$ fois sur le processeur P_j. A chaque exécution d'une règle $R_1|\mathcal{F}$ sur P_j, la règle $R_2|\mathcal{F}$ est potentiellement exécutable une fois. Au total, à partir d'une configuration C_n où tous les processeurs de niveau inférieur ou égal à n n'exécutent plus aucune action, au plus $2 \times \delta^-(j)$ actions peuvent être exécutées sur un processeur P_j de niveau $n+1$.

En conséquence, tout processeur du système exécute un nombre fini d'actions dans toute exécution. □

Lemme 4.4 (Convergence) *En partant d'une configuration de $\mathcal{S}$, toute exécution du système atteint une configuration de l'ensemble $\mathcal{L}_{\mathcal{F}}$.*

Preuve: Nous définissons une fonction variante $\mathcal{M}$ qui renvoie une valeur entière positive ou nulle. Nous montrons que $\mathcal{M}$ est strictement décroissante à chaque exécution d'une règle par un nœud. De plus, quand $\mathcal{M}$ retourne 0 pour tout niveau, le système est dans une configuration de $\mathcal{L}_{\mathcal{F}}$. Soit δ le degré maximum sortant de chacun des nœuds dans le DAG. Alors la fonction $\mathcal{M}$ est définie pour un ensemble de nœuds N_i particulier comme suit :

$$\mathcal{M}(N_i) \equiv \left\{ \sum_{j \in N_i} \left(\phi_{M_j[\delta^-(j)+1] \neq \mathcal{F}\left(M_j[0..\delta^-(j)]\right)} + \sum_{k \in \Gamma_{\mathcal{G}}^{+1}(j)} 2.\phi_{M_j[Ind(k)] \neq M_k[\delta^-(k)+1]} \right) \right\}$$

où ϕ_A est la *fonction caractéristique* de la condition A de telle sorte que ϕ_A retourne 1 si A est *vraie* et 0 sinon. Par exemple, si pour un $j \in N_i$, il existe un $k \in \Gamma_{\mathcal{G}}^{+1}(j)$ tel que $M_j[Ind(k)] \neq M_k[\delta^-(k)+1]$, alors $\phi_{M_j[Ind(k)]\neq M_k[\delta^-(k)+1]}$ renvoie 1. La somme

$$\sum_{k\in\Gamma_{\mathcal{G}}^{+1}(j)} 2 \times \phi_{M_j[Ind(k)]\neq M_k[\delta^-(k)+1]}$$

peut être interprétée comme égale à "deux fois le nombre des descendants immédiats k de j tels que $M_k[\delta^-(k)+1]$ ne correspond pas à la vue qu'en a le processeur P_j".

L'assertion suivante prouve notre lemme

Assertion : La propriété $R(k) \equiv \forall e \in \mathcal{E}, (e = c_1, c_2, \ldots), \exists i \geq 1, \forall j \geq i, \mathcal{M}(N_k) = 0$ dans c_j est *vraie*.

Preuve (par induction sur le niveau des nœuds) :

Cas de Base : Afin de prouver $R(0)$, nous devons montrer que

$$\forall e \in \mathcal{E}, (e = c_1, c_2, \ldots), \exists i \geq 1, \forall j \geq i, \mathcal{M}(N_0) = 0$$

dans c_j. L'ensemble N_0 contient l'ensemble des nœuds source qui n'ont aucun ascendant immédiat. Donc

$$\forall i \in N_0, |\Gamma_{\mathcal{G}}^{+1}(i)| = 0 et \mathcal{M}(N_0) = \sum_{j\in N_0} \phi_{M_j[\delta^-(j)+1]\neq\mathcal{F}\left(M_j[0..\delta^-(j)]\right)}$$

La condition de la fonction caractéristique ϕ est précisément la garde de l'unique règle $R_2|\mathcal{F}_i$ applicable à un nœud source. Cette règle ne peut être appliquée qu'une seule fois sur chacun des nœuds source, et chacune de ces applications fait décroître $\mathcal{M}(N_0)$ de 1. Quand aucune règle n'est applicable sur un nœud source, $\mathcal{M}(N_0) = 0$. Puisque les nœuds de N_0 n'ont pas d'ascendants immédiats, aucune autre règle ne peut être appliquée par un nœud de N_0. Par suite, $\mathcal{M}(N_0)$ reste égal à 0 dans toute exécution à partir du moment où il devient égal à 0. En conséquence, $R(0)$ est *vraie*.

Etape d'induction : Supposons qu'il existe un $k \leq h-1$ tel que $\forall j$ $(0 \leq j \leq k)$, $R(j)$ est *vraie*, où h est la hauteur du *DAG*. Nous devons prouver que $R(k+1)$ est également *vraie*. D'après l'hypothèse d'induction, en partant d'une configuration initiale arbitraire, le système finit par atteindre une configuration c_k, où $\forall j$ $(0 \leq j \leq k)$, $\mathcal{M}(N_j) = 0$ et reste égal à 0 dans toutes les exécutions à partir de c_k. Considérons maintenant les nœuds de l'ensemble N_{k+1}, dont les prédécesseurs directs sont de niveau $\leq k$. Les prédécesseurs directs ne peuvent appliquer aucune règle puisque $\forall j$ $(0 \leq j \leq k)$, $\mathcal{M}(N_j) = 0$. Considérons chacun des nœuds $m \in N_{k+1}$ lisant les variables $M_i[\delta^-(i)+1]$ des ascendants directs i de m. Comme aucun ascendant de m ne peut exécuter de règle, les variables $M_i[\delta^-(i)+1]$ lues par m restent inchangées à partir de la configuration c_k. Le nœud m peut donc exécuter la règle $R_1|\mathcal{F}_m$ au plus $\delta^-(m)$ fois. La règle $R_2|\mathcal{F}_m$ ne peut être appliquée qu'une fois, a moins que la règle $R_1|\mathcal{F}_m$ n'ait été exécutée entre temps. Par la définition de $\mathcal{M}(N_i)$, il existe une relation directe entre les règles de l'algorithme 1 et les fonctions caractéristiques. Chaque

exécution de la règle $R_2|\mathcal{F}_m$ fait décroître $\mathcal{M}(N_{k+1})$ de 1, et chaque exécution de $R_1|\mathcal{F}_m$ fait décroître $\mathcal{M}(N_{k+1})$ de 1 ou 2 suivant qu'il active la règle $R_2|\mathcal{F}_m$ ou pas. De la sorte, $\mathcal{M}(N_{k+1})$ est strictement décroissante à chaque fois qu'une règle est exécutée par un nœud de N_{k+1}. Fatalement, $\mathcal{M}(N_{k+1})$ devient égal à 0 quand toutes les règles sont désactivées à chaque nœud de N_{k+1}. Puisque les ascendants des nœuds de N_{k+1} ne peuvent changer leur registre, les règles demeurent désactivées et $\mathcal{M}(N_{k+1})$ reste égal à 0 dans chaque exécution à partir du moment où il est égal à 0. Par conséquent $R(k+1)$ est *vraie*. D'où l'assertion.

La valeur de $\mathcal{M}(N_i)$ étant égal à 0 pour chaque niveau i du *DAG*, cela implique que toutes les fonctions caractéristiques utilisées dans la définition de $\mathcal{M}(N_i)$ renvoient 0. Par conséquent, le système atteint une configuration de $\mathcal{L}_{\mathcal{F}}$. □

Théorème 1 (Auto-stabilisation) *Le système $\mathcal{S}$ décrit par l'algorithme 1 est auto-stabilisant sur un graphe orienté acyclique pour toute instance de $\mathcal{F}$.*

Preuve: Conséquence des lemmes 4.2 et 4.4. □

4.4.3 Cas Général

Considérons maintenant le cas où le graphe de communication $\mathcal{G}$ du système réparti décrit par l'algorithme 1 peut contenir des cycles. On s'aperçoit aisément que $\mathcal{F}_v$ ne peut plus être une fonction quelconque et le système rester auto-stabilisant.

Cependant, nous montrons que si l'algorithme 1 est instancié avec un r-opérateur n-aire strictement idempotent , alors l'algorithme résultant est auto-stabilisant pour la spécification attendue (voir section 4.4.3), tandis que si $\mathcal{F}_v$ est instanciée avec un r-opérateur n-aire qui n'est pas strictement idempotent, alors l'algorithme résultant n'est pas auto-stabilisant (voir section 4.4.3).

Dans toute cette section, nous supposons que notre algorithme est instancié avec un r-opérateur n-aire comme $\mathcal{F}_v$ sur chaque processeur P_v. Bien qu'il ne soit pas requis que ces opérateurs soient les mêmes pour chaque nœud, nous supposons qu'ils sont tous basés sur le même s-opérateur $\oplus$, de telle sorte que la relation d'ordre définie par $\oplus$ est une relation d'ordre total (hypothèse 4.1). En particulier, ceci implique que pour tout ensemble de données $\{x_0, \ldots, x_k\}$, nous ayons $\oplus\{x_0, \ldots, x_k\} \in \{x_0, \ldots, x_k\}$.

Notation 1 *Nous notons r_v^i la $i^{ème}$ r-fonction du processeur P_v, pour i dans l'ensemble $\{1, \ldots, \delta^-(v)\}$. Afin d'abréger nos notations, nous admettons également que pour tout nœud v, il existe une r-fonction r_v^0 correspondant à la donnée initiale, en définissant $\mathsf{r}_v^0 \equiv \mathsf{Id}$.*

Notation 2 *Soit P un chemin $(u_0, u_1)\ldots(u_{n-1}, u_n)$ dans $\mathcal{G}$. A chacun de ses arcs (u_i, u_{i+1}) correspond la r-fonction $\mathsf{r}_{u_i}^{Ind(u_{i-1})}$. La composition de ces r-fonctions le long du chemin P est un r-chemin et est dénoté par $\mathsf{r}_P = \mathsf{r}_{u_n}^{Ind(u_{n-1})} \circ \ldots \circ \mathsf{r}_{u_1}^{Ind(u_0)}$.*

La Stricte Idempotence est Suffisante à l'Auto-stabilisation

La preuve de correction découle du schéma suivant :
(1) Nous prouvons que toute exécution atteint une configuration où aucune garde n'est vraie (lemme 4.5),
(2) Nous prouvons que dans toute configuration où aucune garde n'est vraie, tous les nœuds du graphe $\mathcal{G}$ sont légitimes suivant la définition donnée section 4.4.3 (lemme 4.14).

Configurations Légitimes Nous définissons la propriété de légitimité pour les nœuds à travers un prédicat de légitimité sur les variables de sortie. Nous dénotons par $\mathcal{L}(x)$ la valeur légitime de la variable x, avec :

$$\mathcal{L}\left(M_v[\delta^-(v)+1]\right) = \oplus \left\{ \begin{array}{l} \mathsf{r}_P(M_u[0]), \\ u \in \Gamma^-_{\mathcal{G}}(v), \\ P \text{ un chemin élémentaire de } u \text{ à } v \end{array} \right\}$$

Ainsi, pour tout nœud v, si $M_v[\delta^-(v)+1] = \mathcal{L}\left(M_v[\delta^-(v)+1]\right)$ la variable de sortie $M_v[\delta^-(v)+1]$ est *légitime*. Une variable d'entrée $M_v[Ind(u)]$ est légitime si $M_v[Ind(u)] = \mathcal{L}\left(M_u[\delta^-(u)+1]\right)$. Pour tout nœud v, la donnée initiale $M_v[0]$ est légitime dans toute configuration, puisque la donnée est stockée en ROM. Un processeur P_v est légitime si sa variable de sortie est légitime. Une configuration est légitime si et seulement si tous les processeurs sont légitimes dans cette configuration : l'ensemble $\mathcal{L}$ des configurations légitimes de $\mathcal{S}$ est défini comme $\forall l \in \mathcal{L}$, $\forall v \in \mathcal{G}$, v est légitime dans l.

Silence

Lemme 4.5 (Silence) *Toute exécution de l'algorithme 1 instancié avec un r-opérateur strictement idempotent possède un journal silencieux.*

Preuve: Nous voulons prouver que, partant d'une configuration initiale arbitraire c_1, toute exécution atteint au bout d'un temps fini une configuration où aucune règle (ni $R_1|\mathcal{F}_v$, ni $R_2|\mathcal{F}_v$) n'est activable.

Dans chaque configuration initiale c_1, il existe un nombre fini de données dans le réseau, soit stockées en ROM, soit dans une variable d'entrée $M_v[i]$, $1 \leq i \leq \delta^-(v)$, pour un nœud $v \in \mathcal{G}$, soit dans une variable de sortie $M_v[\delta^-(v)+1]$. Sans perte de généralité, nous pouvons considérer que les résultats calculés par les différents processeurs sont stockés sous la forme d'une expression formelle où les r-fonctions ne sont pas calculées. Par exemple, si le processeur P_v a la valeur x_0 dans $M_v[0]$ et $x_1, \ldots, x_{\delta^-(v)}$ dans $M_v[1..\delta^-(v)]$, alors l'information stockée dans $M_v[\delta^-(v)+1]$ après application de la règle $R_2|\mathcal{F}_v$ est $\mathsf{r}_v^i(M_v[i])$, pour un i dans $\{1, \ldots \delta^-(v)\}$, où

$$\mathsf{r}_v^i\left(M_v[i]\right) = M_v[0] \oplus \mathsf{r}_v^1\left(M_v[1]\right) \oplus \cdots \oplus \mathsf{r}_v^{\delta^-(v)}\left(M_v[\delta^-(v)]\right)$$

Considérons maintenant les mouvements de données initiales pendant une exécution du système.

Chaque exécution de la règle de copie $R_1|\mathcal{F}_v$ déplace la donnée r-augmentée de la variable de sortie $M_u[\delta^-(u)+1]$ de chaque ascendant direct u de v vers les variables d'entrée $M_v[Ind(u)]$ de v. Ceci écrase l'expression précédemment stockée dans ces variables d'entrée. Cette exécution d'une règle de copie peut être interprétée comme un *mouvement horizontal* entre deux nœuds voisins.

Chaque exécution de la règle de calcul $R_2|\mathcal{F}_v$ déplace la variable r-augmentée stockée dans $M_v[i]$ (pour u unique $i \in \{0,\ldots,\delta^-(v)\}$, dans une nouvelle expression $\mathsf{r}_v^i(M_v[i])$ stockée dans $M_v[\delta^-(v)+1]$. Ceci écrase l'expression précédement stockée dans $M_v[\delta^-(v)+1]$. Cette exécution d'une règle de calcul peut être interprètée comme un *mouvement vertical* à l'intérieur d'un même nœud.

Par construction de l'algorithme, les deux propriétés suivantes sont vérifiées dans chaque exécution sur chaque processeur P_v : (i) entre deux exécutions successives de la règle $R_2|\mathcal{F}_v$, au moins une règle $R_1|\mathcal{F}_v$ est exécutée pour un ascendant direct de v, (ii) entre deux exécutions de la règle $R_1|\mathcal{F}_v$ pour un ascendant direct u de v, la règle $R_2|\mathcal{F}_u$ est exécutée sur P_u. Puisque le graphe de communication est fini, il suffit maintenant de prouver que soit le processeur P_v ne peut exécuter indéfiniment la règle $R_1|\mathcal{F}_v$, soit le processeur P_v ne peut exécuter indéfiniment la règle $R_2|\mathcal{F}_v$. Nous choisissons la deuxième approche.

Supposons qu'il existe un processeur P_v de $\mathcal{S}$ qui exécute $R_2|\mathcal{F}_v$ infiniment souvent au cours d'une exécution de $\mathcal{S}$. Initialement, il n'existe dans le réseau qu'un nombre fini de valeurs, donc il existe une valeur ω qui est déplacée verticalement par l'exécution de la règle $R_2|\mathcal{F}_v$ infiniment souvent sur le processeur P_v. Par hypothèse, toute r-fonction r vérifie, pour tout $y \in \mathbb{S}$, $y \prec_\oplus \mathsf{r}(y)$. D'après l'hypothèse 4.2, il existe une configuration c telle que l'une des deux propositions suivante est vérifiée : (i) $M_v[0] \prec_\oplus \mathsf{r}(\ldots(\omega)\ldots)$ ou (ii) $M_v[0] = \mathsf{r}(\ldots(\omega)\ldots) = e_\oplus$, l'élément maximum de $\mathbb{S}$.

1. Dans le premier cas, le résultat d'un calcul local est inférieur ou égal (au sens de $\oplus$) à $M_v[0]$, qui à son tour est différent de $\mathsf{r}(\ldots(\omega)\ldots)$. Dès lors, aucun mouvement vertical de ω ne peut survenir sur P_v, ce qui contredit l'hypothèse.
2. Dans le deuxième cas, où $\mathsf{r}(\ldots(\omega)\ldots) = M_v[0] = e_\oplus$, toute application ultérieure d'une r-fonction laisserait le résultat inchangé (par la proposition 4.5). Dès lors, aucun mouvement vertical de ω ne peut survenir sur P_v, ce qui contredit également l'hypothèse.

Dans chacun des deux cas possibles, l'hypothèse qu'il n'existe un processeur P_v qui exécute la règle $R_2|\mathcal{F}_v$ infiniment souvent est contredite, ce qui prouve que la règle $R_2|\mathcal{F}_v$ est exécutée seulement un nombre fini de fois dans le système $\mathcal{S}$. D'après la stricte alternance des règles $R_2|\mathcal{F}_v$ et $R_1|\mathcal{F}_v$ sur un même processeur P_v, nous pouvons conclure qu'il n'existe aucune exécution infinie (au sens où des règles restent activables) dans le système réparti que nous considérons. □

Correction Nous prouvons maintenant que dans chaque configuration où aucune règle n'est activable, tous les nœuds du système sont légitimes. Puisque nous avons supposé que le s-opérateur $\oplus$ définit un ordre total, il est possible, dans toute configuration du système, de trier les valeurs des variables d'entrée de tous les nœuds du système $\mathcal{S}$.

Définition 4.7 (Λ^c) *Soit $c \in \mathcal{C}$, Λ^c est l'ensemble* trié *de toutes les variables d'entrée qui apparaissent dans c.*

En utilisant la relation d'ordre total $\prec_\oplus$induite par $\oplus$, $\Lambda^c = \{\lambda_0^c, \lambda_1^c, \ldots, \lambda_{k_c}^c\}$ avec $\lambda_i^c \prec_\oplus \lambda_j^c$ pour $i < j$. Pour tout $i \in \{0..k_c\}$, λ_i^c dénote le $i^{\text{ème}}$ élément de Λ^c.

De plus, dans toute configuration, il est possible de répartir les processeurs du système en ensembles distincts. Informellement, la partition des processeurs de $\mathcal{S}$ en ensembles Ψ_n^c se fait suivant la valeur des variables légitimes. Dans cette partition, Ψ_0^c comprend les processeurs qui ont la plus petite variable r-augmentée λ_0^c, Ψ_1^c comprend les processeurs qui ont la plus petite variable r-augmentée suivante $\lambda_1^c \in \Lambda^c \setminus \{\lambda_0^c\}$, et ainsi de suite. Puisque tous les processeurs possèdent au moins une variable légitime (la donnée initiale $M_v[0]$ est placée en ROM), nous obtenons une partition des processeurs de $\mathcal{S}$.

Définition 4.8 (Ψ_n^c) *Soit c une configuration de $\mathcal{S}$, Ψ_n^c est l'ensemble des processeurs du système qui vérifient le critère suivant :*

$$\Psi_n^c = \left\{ P_v \in \mathcal{S} \wedge \begin{array}{l} \exists i, 0 \leq i \leq \delta^-(v), \\ M_v[i] = \mathcal{L}(M_v[i]), \\ \mathsf{r}_v^i(M_v[i]) = \lambda_n^c, \\ \nexists n' < n, P_v \in \Psi_{n'}^c \end{array} \right\}$$

Lemme 4.6 *Dans toute configuration $c \in \mathcal{C}$ où aucune règle n'est activable, il n'existe aucune variable de sortie légitime plus petite que λ_0^c.*

Preuve: Toute variable légitime est égale à une expression construite avec des données initiales augmentées par des r-chemins strictement idempotents. Supposons qu'il existe une configuration $c \in \mathcal{C}$ où aucune règle n'est applicable et telle que :

$$\mathcal{L}\left(M_v[\delta^-(v)+1]\right) \prec_\oplus \lambda_0^c$$

Alors, nous aurions :

$$\oplus \left\{ \mathsf{r}_P\left(M_u[0]\right), u \in \Gamma_{\mathcal{G}}^-(v), P \text{ un chemin élémentaire de } u \text{ à } v \right\} \prec_\oplus \lambda_0^c$$

et il existerait $u \in \Gamma_{\mathcal{G}}^-(v)$ et un r-chemin r_P vérifiant :

$$M_u[0] \prec_\oplus \mathsf{r}_P\left(M_u[0]\right) \prec_\oplus \lambda_0^c$$

ce qui est impossible, par définition de λ_0^c. □

Lemme 4.7 *Tout processeur $P_v \in \Psi_0^c$ vérifie $\mathcal{L}(M_v[\delta^-(v)+1]) = \mathcal{L}(M_v[0]) = M_v[0] = \lambda_0^c$.*

Preuve: Nous avons :

$$\mathcal{L}(M_v[\delta^-(v)+1]) = \oplus \left\{ \mathsf{r}_v^i\left(\mathcal{L}\left(M_v[i]\right)\right), 0 \leq i \leq \delta^-(v) \right\}$$

et, puisque $P_v \in \Psi_0^c$,

$$\mathcal{L}(M_v[\delta^-(v)+1]) \preceq_\oplus \lambda_0^c$$

D'après le lemme 4.6, aucune variable de sortie légitime ne peut être inférieure à λ_0^c, ce qui signifie que :

$$\mathcal{L}(M_v[\delta^-(v)+1]) = \lambda_0^c$$

Supposons qu'il existe un entier q, $1 \leq q \leq \delta^-(v)$ tel que :

$$\mathcal{L}(M_v[\delta^-(v)+1]) = \mathsf{r}_v^q\left(\mathcal{L}(M_v[q])\right)$$

Par hypothèse, toutes les r-fonction sont strictement idempotentes. Nous obtenons :

$$\begin{array}{rcl} \mathcal{L}(M_v[q]) & \prec & {}_\oplus\mathsf{r}_v^q\left(\mathcal{L}(M_v[q])\right) \\ & \prec & {}_\oplus\lambda_0^c \end{array}$$

ce qui est impossible, par définition de λ_0^c. Donc, $\mathcal{L}(M_v[\delta^-(v)+1]) = \mathcal{L}(M_v[0]) = M_v[0]$. □

Pour une configuration $c \in \mathcal{C}$ donnée, nous considérons la proposition $P_{n,k}^c$, qui a un usage technique dans les preuves par induction qui suivent, et qui se décrit formellement comme suit :

$$\begin{array}{ll} P_{n,k}^c : & \exists v \in \mathcal{G}, \exists n_v \geq n, P_v \in \Psi_{n_v}^c, \\ & \exists u_1 \in \Gamma_{\mathcal{G}}^{-1}(v), \ldots, u_k \in \Gamma_{\mathcal{G}}^{-k}(v), \\ & \forall i, 1 \leq i \leq k, \exists n_i \geq n, P_{u_i} \in \Psi_{n_i}^c, \\ & (u_k, u_{k-1}), \ldots, (u_2, u_1), (u_1, v) \in \mathcal{G} \\ & M_{u_k}[\delta^-(u_k)+1] \prec_\oplus \ldots \prec_\oplus M_{u_1}[\delta^-(u_1)+1] \prec_\oplus \lambda_n^c \end{array}$$

Informellement, $P_{n,k}^c$ affirme que dans la configuration $c \in \mathcal{C}$, il existe un chemin élémentaire de longueur k dont chacun des processeurs est non-légitime.

En effet, si $P_v \in \Psi_{n_v}^c$, alors la plus petite variable d'entrée r-augmentée de P_v est égale à λ_n^c. Si la variable de sortie est plus petite que λ_n^c, alors cette valeur a été obtenue à partir d'une autre variable d'entrée r-augmentée, elle-même inférieure à λ_n^c. Par conséquent, cette autre variable d'entrée ne peut être légitime, et par suite P_v ne peut être légitime lui non plus. En utilisant le même raisonnement, tous les processeurs présents sur le chemin sont non-légitimes.

Nous souhaitons prouver que dans une configuration c_t où aucune règle n'est applicable, tous les processeurs sont légitimes, ce qui revient à dire que la configuration elle-même est légitime. La preuve se fait par récurrence sur les ensembles Ψ_n^c, qui définissent une partition des processeurs de $\mathcal{A}$.

La récurrence est divisée en deux parties, le cas de base (lemme 4.10) et l'étape de récurrence (lemme 4.13). La première partie (le cas de base) est prouvé par récurrence sur $P_{0,k}^{c_t}$ (le cas de base est prouvé par le lemme 4.8, l'étape de récurrence par le lemme 4.9 et la conclusion par le lemme 4.10). De manière similaire, l'étape d'induction est prouvée par récurrence sur $P_{n,k}^{c_t}$ (le cas de base est prouvé par le lemme 4.11, l'étape de récurrence par

le lemme 4.12 et la conclusion par le lemme 4.13). Pour chacune des deux parties, la preuve est basée sur le fait que puisque le graphe de communication est fini, il ne peut exister de chemin élémentaire infini de processeurs non légitimes.

Dans la suite de la section $\mathcal{C}_t \subset \mathcal{C}$ désigne l'ensemble des configurations où aucune règle n'est applicable.

Lemme 4.8 *Dans toute configuration $c_t \in \mathcal{C}_t$ où il existe un processeur non légitime $P_v \in \Psi_0^{c_t}$, $P_{0,1}^{c_t}$ est vraie.*

Preuve: La proposition $P_{0,1}^{c_t}$ s'écrit comme suit :

$$\begin{array}{ll} P_{0,1}^{c_t} : & \exists v \in \mathcal{G}, \exists n_v \geq 0, P_v \in \Psi_{n_v}^{c_t}, \\ & \exists u_1 \in \Gamma_{\mathcal{G}}^{-1}(v), \\ & \exists n_1 \geq 0, P_{u_1} \in \Psi_{n_1}^{c_t}, \\ & (u_1, v) \in \mathcal{G}, \\ & M_{u_1}[\delta^-(u_1)+1] \prec_\oplus \lambda_0^{c_t} \end{array}$$

Considérons un processeur $P_v \in \Psi_0^{c_t}$. D'après le lemme 4.7, nous avons

$$\mathcal{L}(M_v[\delta^-(v)+1]) = M_v[0] = \lambda_0^{c_t}$$

Supposons que dans la configuration c_t, le processeur P_v soit non-légitime :

$$M_v[\delta^-(v)+1] \neq \mathcal{L}\left(M_v[\delta^-(v)+1]\right)$$

Dans la configuration c_t, aucune règle $R_2|\mathcal{F}_v$ n'est applicable, soit formellement :

$$\begin{array}{lll} M_v[\delta^-(v)+1] & = & \oplus\left\{\mathsf{r}_v^i\left(M_v[i]\right), 0 \leq i \leq \delta^-(v)\right\} \\ & \prec_\oplus & M_v[0] \quad \text{(ne peut être supérieur, si égal alors légitime)} \\ & \prec_\oplus & \lambda_0^{c_t} \end{array}$$

Donc le processeur P_v possède au moins une variable d'entrée r-augmentée strictement inférieure à $\lambda_0^{c_t}$:

$$\exists u_1 \in \Gamma_{\mathcal{G}}^-(v), \mathsf{r}_v^{Ind(u_1)}\left(M_v[Ind(u_1)]\right) \prec_\oplus \lambda_0^{c_t}$$

Puisque toutes les r-fonctions sont strictement idempotentes, nous obtenons :

$$\exists u_1 \in \Gamma_{\mathcal{G}}^-(v), M_v[Ind(u_1)] \prec_\oplus \mathsf{r}_v^{Ind(u_1)}\left(M_v[Ind(u_1)]\right) \prec_\oplus \lambda_0^{c_t}$$

Dans la configuration c_t, aucune règle $R_1|\mathcal{F}_v$ n'est applicable, soit formellement :

$$\exists u_1 \in \Gamma_{\mathcal{G}}^-(v), M_{u_1}[\delta^-(u_1)+1] = M_v[Ind(u_1)] \prec_\oplus \mathsf{r}_v^{Ind(u_1)}(Ind(u_1)) \prec_\oplus \lambda_0^{c_t}$$

Notons que (u_1, v) est un chemin de longueur 1. De plus, P_{u_1} est non-légitime (par le lemme 4.7). Nous avons donc prouvé que s'il existe une configuration $c_t \in \mathcal{C}_t$ dans laquelle un processeur P_v est non-légitime, la proposition $P_{0,1}^{c_t}$ est vraie, signifiant qu'il existe un chemin élémentaire de longueur 1 de processeurs non-légitimes. □

Lemme 4.9 *Dans toute configuration c_t où il existe un processeur non-légitime $P_v \in \Psi_0^{c_t}$, $P_{0,k}^{c_t}$ est vraie, pour tout $k \in \mathbb{N}$.*

Preuve: Prouvons par récurrence sur k la proposition.

Cas de Base. Le cas de base est prouvé par le lemme 4.8.

Etape de Récurrence. Supposons qu'il existe une configuration $c_t \in \mathcal{C}_t$ et un entier $k > 1$ tel que $P_{0,k-1}^{c_t}$ soit vraie :

$$\begin{array}{rl} P_{0,k-1}^{c_t} : & \exists v \in \mathcal{G}, \exists n_v \geq 0, P_v \in \Psi_{n_v}^{c_t}, \\ & \exists u_1 \in \Gamma_{\mathcal{G}}^{-1}(v), \ldots, u_{k-1} \in \Gamma_{\mathcal{G}}^{-(k-1)}(v), \\ & \forall i, 1 \leq i < k, \exists n_i \geq 0, P_{u_i} \in \Psi_{n_i}^{c_t}, \\ & (u_{k-1}, u_{k-2}), \ldots, (u_2, u_1), (u_1, v) \in \mathcal{G}, \\ & M_{u_{k-1}}[\delta^-(u_{k-1}) + 1] \prec_\oplus \ldots \prec_\oplus M_{u_1}[\delta^-(u_1) + 1] \prec_\oplus \lambda_0^{c_t} \end{array}$$

Puisque nous avons $c_t \in \mathcal{C}_t$, chacune des variables de sortie $M_{u_{k-1}}[\delta^-(u_{k-1})+1]$ de chacun des processeurs $P_{u_{k-1}}$ de $\Gamma_{\mathcal{G}}^{-(k-1)}(v)$ est égale à la donnée initiale $M_{u_{k-1}}[0]$ où à l'une de ses variables d'entrée r-augmentées, $\mathsf{r}_v^i\left(M_{u_{k-1}}[i]\right)$, $1 \leq i \leq \delta^-(u_{k-1})$. Or, par définition, aucune donnée initiale ne peut être inférieure à $\lambda_0^{c_t}$. Nous pouvons donc écrire :

$$\begin{array}{l} \exists u_k \in \Gamma_{\mathcal{G}}^{-k}(v), (u_k, u_{k-1}) \in \mathcal{G}, \\ \quad \mathsf{r}_{u_{k-1}}^{Ind(u_k)}\left(M_{u_{k-1}}[Ind(u_k)]\right) \end{array} \begin{array}{ll} = & M_{u_{k-1}}[\delta^-(u_{k-1}) + 1] \\ \prec_\oplus & \ldots \\ \prec_\oplus & M_{u_1}[\delta^-(u_1) + 1] \\ \prec_\oplus & \lambda_0^{c_t} \end{array}$$

Puisque $c_t \in \mathcal{C}_t$, aucune règle $R_1|\mathcal{F}_u$ n'est applicable pour $u \in \Gamma_{\mathcal{G}}^{-(k-1)}(v)$, et nous avons :

$$\begin{array}{l} \exists u_k \in \Gamma_{\mathcal{G}}^{-k}(v), (u_k, u_{k-1}) \in \mathcal{G}, \\ M_{u_k}[\delta^-(u_k) + 1] = M_{u_{k-1}}[Ind(u_k)], \\ \quad \mathsf{r}_{u_{k-1}}^{Ind(u_k)}\left(M_{u_{k-1}}[Ind(u_k)]\right) \end{array} \begin{array}{ll} = & M_{u_{k-1}}[\delta^-(u_{k-1}) + 1] \\ \prec_\oplus & \ldots \\ \prec_\oplus & M_{u_1}[\delta^-(u_1) + 1] \\ \prec_\oplus & \lambda_0^{c_t} \end{array}$$

Comme les r-fonctions sont strictement idempotentes, nous avons :

$$\begin{array}{l} \exists u_k \in \Gamma_{\mathcal{G}}^{-k}(v), (u_k, u_{k-1}) \in \mathcal{G}, \\ \quad M_{u_k}[\delta^-(u_k) + 1] \end{array} \begin{array}{ll} \prec_\oplus & M_{u_{k-1}}[\delta^-(u_{k-1}) + 1] \\ \prec_\oplus & \ldots \\ \prec_\oplus & M_{u_1}[\delta^-(u_1) + 1] \\ \prec_\oplus & \lambda_0^{c_t} \end{array}$$

Ce qui est une réécriture de la proposition $P_{0,k}^{c_t}$.

Conclusion. Nous avons montré par récurrence que sous les hypothèses prises, la proposition $P_{0,k}^{c_t}$ est vraie pour tout $k \in \mathbb{N} \setminus \{0\}$. □

Lemme 4.10 *Dans chaque configuration $c_t \in \mathcal{C}_t$, aucun processeur de l'ensemble $\Psi_0^{c_t}$ n'est non-légitime.*

Preuve: D'après le lemme 4.9, dans chaque configuration $c_t \in \mathcal{C}_t$, $P_{0,k}^{c_t}$ est vraie pour tout $k \in \mathbb{N} \setminus \{0\}$. Puisque $P_{0,k}^{c_t}$ ne peut être vraie pour k supérieur au nombre de nœuds du graphe de communication, aucun processeur de l'ensemble $\Psi_0^{c_t}$ ne peut être non-légitime dans c_t. □

Le lemme 4.10 établit le cas de base pour la récurrence sur les ensembles $\Psi_n^{c_t}$. Nous prouvons maintenant l'étape d'induction au moyen des lemmes 4.11 et 4.12.

Lemme 4.11 *Dans toute configuration $c_t \in \mathcal{C}_t$ où tous les processeurs $P_u \in \Psi_0^{c_t} \cup \ldots \cup \Psi_{n-1}^{c_t}$ sont légitimes et où il existe un processeur non légitime $P_v \in \Psi_n^{c_t}$, $P_{n,1}^{c_t}$ est vraie pour tout $n \in \mathbb{N}$.*

Preuve: La proposition $P_{n,1}^{c_t}$ s'écrit comme suit :

$$\begin{array}{ll} P_{n,1}^{c_t} : & \exists v \in \mathcal{G}, \exists n_v \geq n, P_v \in \Psi_{n_v}^{c_t}, \\ & \exists u_1 \in \Gamma_{\mathcal{G}}^{-1}(v), \\ & \exists n_1 \geq n, P_{u_1} \in \Psi_{n_1}^{c_t}, \\ & (u_1, v) \in \mathcal{G}, \\ & M_{u_1}[\delta^-(u_1)+1] \prec_{\oplus} \lambda_n^{c_t} \end{array}$$

Considérons $P_v \in \Psi_n^{c_t}$. Il existe un entier q, $1 \leq q \leq \delta^-(v)$, tel que $M_v[q]$ est légitime et $\mathsf{r}_v^q\left(M_v[q]\right) = \lambda_n^{c_t}$. Par hypothèse, P_v est non-légitime, soit : $M_v[\delta^-(v)+1] \neq \mathcal{L}\left(M_v[\delta^-(v)+1]\right)$.

Dans la configuration c_t, aucune règle $R_2|F_v$ n'est applicable, soit formellement :

$$\begin{array}{lll} M_v[\delta^-(v)+1] & = & \oplus\left\{\mathsf{r}_v^i\left(M_v[i]\right), 0 \leq i \leq \delta^-(v)\right\} \\ & \prec_{\oplus} & \lambda_n^{c_t} \end{array}$$

Supposons que $M_v[\delta^-(v)+1] = M_v[0]$. Alors nous aurions $M_v[0] \prec_{\oplus} \lambda_n^{c_t}$. Or, $M_v[0] = \mathcal{L}(M_v[0])$, et P_v ne serait pas dans $\Psi_n^{c_t}$. Donc nous avons :

$$\begin{array}{lll} \exists u_1 \in \Gamma_{\mathcal{G}}^{-1}(v), \mathsf{r}_v^{Ind(u_1)}\left(M_v[Ind(u_1)]\right) & = & M_v[\delta^-(v)+1] \\ & \prec_{\oplus} & \lambda_n^{c_t} \end{array}$$

Puisque toutes les r-fonctions sont strictement idempotentes, nous avons :

$$\exists u_1 \in \Gamma_{\mathcal{G}}^{-1}(v), M_v[Ind(u_1)] \prec_{\oplus} \mathsf{r}_v^{Ind(u_1)}\left(M_v[Ind(u_1)]\right) \prec_{\oplus} \lambda_n^{c_t}$$

Dans la configuration c_t, aucune règle $R_1|F_v$ n'est applicable, soit formellement :

$$\exists u_1 \in \Gamma_{\mathcal{G}}^{-}(v), M_{u_1}[\delta^-(u_1)+1] = M_v[Ind(u_1)] \prec_{\oplus} \mathsf{r}_v^{Ind(u_1)}(Ind(u_1)) \prec_{\oplus} \lambda_n^{c_t}$$

Puisque $M_v[Ind(u_1)] \prec_{\oplus} \lambda_n^{c_t}$ et $P_v \in \Psi_n^{c_t}$, la variable $M_v[Ind(u_1)]$ ne peut être légitime. Donc $M_{u_1}[\delta^-(v)+1]$ n'est pas légitime et le processeur P_{u_1} n'est pas légitime non plus, ce qui signifie que $P_{u_1} \in \Psi_{n_1}^{c_t}$ avec $n_1 \geq n$.

Nous avons prouvé que s'il existe une configuration $c_t \in \mathcal{C}_t$ où un processeur P_v n'est pas légitime, la proposition $P_{n,1}^{c_t}$ est vraie, signifiant qu'il existe un chemin élémentaire de longueur 1 de processeurs non-légitimes. □

Lemme 4.12 *Dans toute configuration $c_t \in \mathcal{C}_t$ où tout processeur $P_u \in \Psi_0^{c_t} \cup \ldots \cup \Psi_{n-1}^{c_t}$ est légitime et où il existe un processeur non légitime $P_v \in \Psi_n^{c_t}$, $P_{n,k}^{c_t}$ est vraie pour tout $n, k \in \mathbb{N}$.*

Preuve: Prouvons par récurrence sur k que la propriété est vérifiée.

Case de Base. Le cas de base est prouvé par le lemme 4.11.

Etape d'Induction. Supposons qu'il existe une configuration $c_t \in \mathcal{C}_t$ telle que $P_{n,k-1}^{c_t}$ soit vraie :

$$\begin{array}{rl} P_{n,k-1}^{c_t} : & \exists v \in \mathcal{G}, \exists n_v \geq n, P_v \in \Psi_{n_v}^{c_t}, \\ & \exists u_1 \in \Gamma_{\mathcal{G}}^{-}(v), \ldots, u_{k-1} \in \Gamma_{\mathcal{G}}^{-(k-1)}(v), \\ & \forall i, 1 \leq i < k, \exists n_i \geq n, P_{u_i} \in \Psi_{n_i}^{c_t}, \\ & (u_{k-1}, u_{k-2}), \ldots, (u_2, u_1), (u_1, v) \in \mathcal{G} \\ & M_{u_{k-1}}[\delta^-(u_{k-1}) + 1] \prec_\oplus \ldots \prec_\oplus M_{u_1}[\delta^-(u_1) + 1] \prec_\oplus \lambda_n^{c_t} \end{array}$$

Puisque $c_t \in \mathcal{C}_t$, chacune des variables de sortie $M_{u_{k-1}}[\delta^-(u_{k-1}) + 1]$ de chacun des processeurs $P_{u_{k-1}}$ de $\Gamma_{\mathcal{G}}^{-(k-1)}(v)$ est égale à la donnée initiale $M_{u_{k-1}}[0]$ où à l'une de ses variables d'entrée r-augmentées, $\mathsf{r}_v^i\left(M_{u_{k-1}}[i]\right)$, $1 \leq i \leq \delta^-(u_{k-1})$. Aucune variable légitime r-augmentée n'est inférieure à $\lambda_n^{c_t}$ sur $P_{u_{k-1}}$, sinon on aurait $P_{u_{k-1}} \in \Psi_{n'}^{c_t}$ avec $n' < n$. En particulier, $M_{u_{k-1}}[0] \not\prec_\oplus \lambda_n^{c_t}$. Par conséquent, nous pouvons écrire :

$$\begin{array}{l} \exists u_k \in \Gamma_{\mathcal{G}}^{-k}(v), (u_k, u_{k-1}) \in \mathcal{G}, \\ \mathsf{r}_{u_{k-1}}^{Ind(u_k)}\left(M_{u_{k-1}}[Ind(u_k)]\right) \quad = \quad M_{u_{k-1}}[\delta^-(u_{k-1}) + 1] \\ \qquad\qquad\qquad\qquad\qquad\quad \prec_\oplus \quad \ldots \\ \qquad\qquad\qquad\qquad\qquad\quad \prec_\oplus \quad M_{u_1}[\delta^-(u_1) + 1] \\ \qquad\qquad\qquad\qquad\qquad\quad \prec_\oplus \quad \lambda_n^{c_t} \end{array}$$

Puisque $c_t \in \mathcal{C}_t$, aucune règle $R_1|\mathcal{F}_u$ n'est applicable pour $u \in \Gamma_{\mathcal{G}}^{-(k-1)}(v)$, et nous avons :

$$\begin{array}{l} \exists u_k \in \Gamma_{\mathcal{G}}^{-k}(v), (u_k, u_{k-1}) \in \mathcal{G}, \\ M_{u_k}[\delta^-(u_k) + 1] = M_{u_{k-1}}[Ind(u_k)], \\ \mathsf{r}_{u_{k-1}}^{Ind(u_k)}\left(M_{u_{k-1}}[Ind(u_k)]\right) \quad = \quad M_{u_{k-1}}[\delta^-(u_{k-1}) + 1] \\ \qquad\qquad\qquad\qquad\qquad\quad \prec_\oplus \quad \ldots \\ \qquad\qquad\qquad\qquad\qquad\quad \prec_\oplus \quad M_{u_1}[\delta^-(u_1) + 1] \\ \qquad\qquad\qquad\qquad\qquad\quad \prec_\oplus \quad \lambda_n^{c_t} \end{array}$$

Puisque toutes les r-fonctions sont strictement idempotentes, nous avons :

$$\begin{array}{l} \exists u_k \in \Gamma_{\mathcal{G}}^{-k}(v), (u_k, u_{k-1}) \in \mathcal{G}, \\ M_{u_k}[\delta^-(u_k) + 1] \quad \prec_\oplus \quad M_{u_{k-1}}[\delta^-(u_{k-1}) + 1] \\ \qquad\qquad\qquad\quad \prec_\oplus \quad \ldots \\ \qquad\qquad\qquad\quad \prec_\oplus \quad M_{u_1}[\delta^-(u_1) + 1] \\ \qquad\qquad\qquad\quad \prec_\oplus \quad \lambda_n^{c_t} \end{array}$$

Finalement, $M_{u_{k-1}}[Ind(u_k)]$ n'est pas légitime, sinon $P_{u_{k-1}}$ ne serait pas dans l'ensemble $\Psi_n^{c_t}$ $(M_{u_{k-1}}[Ind(u_k)] \prec_\oplus \lambda_n^{c_t})$. Par conséquent, $M_{u_k}[\delta^-(v) + 1]$ n'est pas légitime et donc $P_{u_k} \in \Psi_{n'}^{c_t}$, avec $n' \geq n$. Ceci est une réécriture de $P_{n,k}^{c_t}$.

Conclusion. Nous avons prouvé par récurrence que sous les hypothèses prises, la proposition $P_{n,k}^{c_t}$ était vraie pour tout $k \in \mathbb{N} \setminus \{0\}$. □

Lemme 4.13 *Pour toute configuration $c_t \in \mathcal{C}_t$, si tout processeur $P_u \in \Psi_0^{c_t} \cup \ldots \cup \Psi_{n-1}^{c_t}$ est légitime, alors tout processeur $P_v \in \Psi_n^{c_t}$ est légitime.*

Preuve: Le lemme 4.12 prouve que s'il existe une configuration $c_t \in \mathcal{C}_t$ où tous les processeurs $P_u \in \Psi_0^{c_t} \cup \ldots \cup \Psi_{n-1}^{c_t}$ sont légitimes et où il existe un processeur non-légitime $P_v \in \Psi_n^{c_t}$, alors $P_{n,k}^{c_t}$ est vraie pour tout $k \in \mathbb{N} \setminus \{0\}$ et pour tout $n \in \mathbb{N}$. Puisque le graphe de communication du système est fini, $P_{n,k}^{c_t}$ ne peut être vraie pour k supérieur au nombre de nœuds du graphe de communication du système. □

Lemme 4.14 (Correction) *Pour toute configuration $c_t \in \mathcal{C}_t$, tous les processeurs de $\mathcal{S}$ sont légitimes.*

Preuve: Prouvons par récurrence sur n que tous ensembles $\Psi_n^{c_t}$, qui définissent une partition de $\mathcal{S}$, ne contiennent que des processeurs légitimes.

Cas de Base. Le cas de base est prouvé par le lemme 4.10.

Etape d'Induction. L'étape d'induction est prouvée par le lemme 4.13.

Conclusion. Tous les processeurs de $\mathcal{S}$ sont légitimes dans c_t. □

Des lemmes 4.5 et 4.14, nous pouvons énoncer la condition suffisante suivante :

Proposition 4.7 (Condition Suffisante) *Si l'algorithme 1 est instancié en utilisant des r-opérateurs n-aires strictement idempotents, alors il est auto-stabilisant.*

Preuve: Nous avons prouvé que toute exécution e de notre algorithme atteignait nécessairement une configuration c_t où aucune règle n'était applicable (lemme 4.5). Par la suite, nous avons prouvé que chacune de ces configurations c_t était légitime (lemme 4.14). En conséquence, l'ensemble $\mathcal{C}_t$ est un attracteur clos pour $\mathcal{C}$, l'ensemble des configurations du système $\mathcal{S}$, donc le système est auto-stabilisant. □

La Stricte Idempotence est Nécessaire à l'Auto-stabilisation

Dans cette section, nous montrons que la propriété de stricte idempotence est nécessaire à l'auto-stabilisation de notre algorithme dans certains systèmes basés sur les r-opérateurs.

Proposition 4.8 (Condition Nécessaire) *Si l'algorithme 1est instancié avec des r-opérateurs n-aires qui ne sont pas strictement idempotents comme $\mathcal{F}_v$, alors le système n'est pas auto-stabilisant.*

Preuve: La preuve se fait en exhibant un graphe de communication particulier et une initialisation particulière tels que le système ne stabilise pas sur une configuration légitime.

Considérons le système réparti constitué d'un unique processeur P_v et dont le graphe de communication est réduit à un nœud v et un arc (v, v), comme présenté dans la figure 4.2.

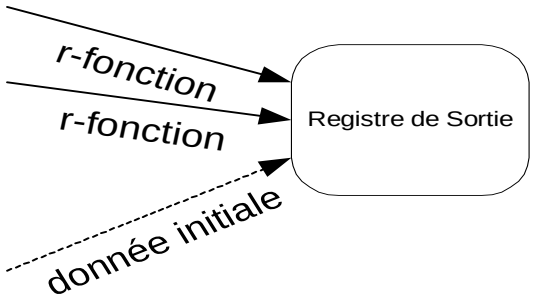

FIG. 4.2 – Si r n'est pas strictement idempotent, le système peut ne pas stabiliser.

Supposons que P_v utilise le r-opérateur binaire non-strictement idempotent $\triangleleft$ défini par $x \triangleleft y \equiv x \oplus \mathsf{r}(y)$ pour tout $x, y \in \mathbb{S}$, où $\oplus$ est un s-opérateur sur $\mathbb{S}$ et r est un homomorphisme de $(\mathbb{S}, \oplus)$ (voir section 4.2.2). Soit $a \in \mathbb{S} \setminus \{e_\oplus\}$ et tel que $a = \mathsf{r}(a)$, contredisant l'hypothèse de stricte idempotence. Considérons maintenant le cas où le processeur P_v possède dans sa donnée initiale $M_v[0]$ une donnée b supérieure à a (au sens de $\preceq_\oplus$, la relation d'ordre total induite par $\oplus$). Supposons de plus que $M_v[1] = b$ à l'origine. En l'absence de nouvelles pannes, les variables $M_v[1]$ et $M_v[2]$ stabilisent sur la valeur b.

Supposons maintenant qu'à la suite de défaillances transitoires, $M_v[2]$ contienne une donnée erronée a. Cette donnée est copiée dans $M_v[1]$, et l'exécution aboutit à une configuration où aucune règle n'est applicable et où P_v est non-légitime (puisque $\mathsf{r}(a) = a$ et $b \oplus a = a$). Donc l'algorithme 1 instancié avec le r-opérateur non-strictement idempotent $\triangleleft$ n'est pas auto-stabilisant. □

Des propositions 4.7 et 4.8, nous pouvons conclure que lorsque nous considérons l'algorithme 1 instancié avec des r-opérateurs comme $\mathcal{F}_v$, le système résultant est auto-stabilisant sur tout graphe orienté si et seulement si les r-opérateurs sont strictement idempotents.

4.4.4 Complexité

Lemme 4.15 (Espace Mémoire) *Chaque processeur $P_v \in \mathcal{A}$ utilise $\delta^-(v) \times \log_2(|\mathbb{S}|)$ bits, où $\mathbb{S}$ est l'ensemble des valeurs des registres, et où $|\mathbb{S}|$ dénote le nombre de ses éléments.*

Preuve: Chaque processeur P_v possède $\delta^-(v) - 1$ variables d'entrée pour contenir la valeur de sortie de chacun de ses ascendants directs, et un registre pour stocker sa variable de sortie. Chacune de ces variables contient une valeur prise dans un ensemble $\mathbb{S}$, et nécessite donc $\log_2(|\mathbb{S}|)$ bits. Notons que la constante stockée en ROM n'est pas prise en compte dans ce calcul. □

Nous traitons séparément le temps de stabilisation dans les graphes sans cycles et le temps de stabilisation dans le cas général.

Théorème 2 (Temps de Stabilisation (DAG)) *Le temps de stabilisation du système $\mathcal{S}$ est en $O(D)$, où D est le diamètre du graphe de communication acyclique $\mathcal{G}$ de $\mathcal{S}$.*

Preuve: Nous supposons qu'en un cycle asynchrone d'exécution, un processeur exécute *toutes* ses actions gardées constamment activables. Comme vu dans la preuve du lemme 4.4, un processeur peut exécuter au plus $2\delta + 1$ actions en un cycle, où δ est le degré sortant du graphe orienté acyclique de communication. En partant d'une configuration arbitraire, en un cycle asynchrone d'exécution, aucune règle n'est activable sur un processeur de l'ensemble N_0 des processeurs de niveau 0. Quand toutes les règles gardées jusqu'au niveau i ne sont plus activables, les entrées des processeurs du niveau $i+1$ ne peuvent plus changer. Donc, en un cycle asynchrone d'exécution supplémentaire, toutes les règles du niveau $i+1$ ne sont plus activables. Après que le plus haut niveau a été atteint, aucune règle n'est activable dans le réseau et le système est stabilisé. Puisque le niveau maximum du graphe de communication est limité par son diamètre D, le système stabilise en un temps $O(D)$. □

Lemme 4.16 (Temps de Stabilisation) *Le temps nécessaire à la stabilisation du système $\mathcal{S}$ est en $O(D + |\mathbb{S}|)$ cycles asynchrones d'exécution, où D est le diamètre du graphe de communication $\mathcal{G}$ de $\mathcal{S}$.*

Preuve: Soit ϕ la fonction retournant l'indice d'un élément de $\mathbb{S}$. Cet indice existe toujours puisque $\mathbb{S}$ est totalement ordonné et que s'il est infini, alors toute suite strictement croissante est non bornée. La signature de ϕ s'énonce comme suit :

$$\begin{array}{rccc} \phi : & \mathbb{S} & \rightarrow & \mathbb{N} \\ & s & \mapsto & \phi(s) \end{array} \qquad \text{et} \qquad s_1 \prec_{\oplus} s_2 \quad \Rightarrow \quad \phi(s_1) < \phi(s_2)$$

Après $O(D)$ cycles asynchrones d'exécution, chaque processeur du réseau a reçu des valeurs r-augmentées en provenance de chacun de ses ancêtres.

Pour chaque processeur P_u, considérons la différence entre l'indice de sa valeur finale (puisque tout journal d'exécution de $\mathcal{S}$ est silencieux) et l'indice de la plus petite valeur initialement reçue (après $O(D)$ cycles asynchrones d'exécution). La plus grande différence possible est $M - m$, où M est l'indice maximum possible dans $\mathbb{S}$ pour une configuration légitime, et m l'indice minimum des valeurs de $\mathbb{S}$. Cette différence est notée d est en $O(|\mathbb{S}|)$.

Pour chaque processeur P_u, considérons également le plus petit et le plus grand (au sens de l'accroissement) r-chemin de P_u à P_u. Soit l la longueur de ce r-chemin. Le plus grand de ces r-chemins augmente l'indice d'une valeur par au moins l.

Dans le pire des cas, il existe un processeur qui possède une variable de sortie incorrecte, indicée par m, une valeur d'entrée correcte indicée par M, de manière qu'il doit attendre que la valeur incorrecte s'accroîsse de $M - m$. Tous les l cycles asynchrones d'exécution au moins, cette valeur incorrecte s'accroît de l. Dans le pire des cas, si $\lfloor \frac{d}{l} \rfloor < \frac{d}{l}$, une autre valeur incorrecte peut encore être inférieure à la valeur correcte, et un cycle peut à nouveau être suivi, induisant un délai supplémentaire de d cycles asynchrones d'exécution.

En tout, après les premiers $O(D)$ cycles asynchrones d'exécution, $\left(\lfloor \frac{d}{l} \rfloor \times l\right) + d = O(d)$ cycles asynchrones d'exécution peuvent s'avérer nécessaires. □

4.5 Applications

4.5.1 Tri Topologique sur des Graphes sans Cycles

Spécification du problème

Chaque processeur P_i possède une liste ordonnée Γ_i, de ses ascendants. Formellement, $\Gamma_i = \{P_j | P_j \text{ est un ascendant de } P_i\}$.

Nous définissons $\mathcal{L}_{\mathcal{T}}$ comme l'ensemble des configurations légitimes pour le problème du tri topologique, où les deux conditions suivantes sont vérifiées :

1. $TS_1 : \forall i \in V$, $P_j \in \Gamma_i$ si et seulement si P_j est un ascendant de P_i
2. $TS_2 : \forall i \in V$, $\forall P_j \in \Gamma_i$, P_j apparaît dans Γ_i avant chacun de ses ascendants.

La condition TS_1 indique que Γ_i contient tous les ascendants de P_i et seulement eux. La condition TS_2 établit que tous les processeurs atteignables sont ordonnés de telle manière qu'un processeur apparaît toujours avant tous ses ascendants. La condition TS_2 définit un *ordre partiel* sur l'ensemble Γ_i, donc l'ensemble des configurations légitimes peut contenir plus d'une configuration (formellement, $|\mathcal{L}_{\mathcal{T}}| \geq 1$).

Algorithme

Nous supposons que chaque processeur possède une liste de couples $\langle P_j, l_j \rangle$, où P_j est un ascendant de P_i et où l_j est son *niveau*. Par ailleurs, chaque processeur dispose de deux fonctions : Max retourne le couple dont le niveau est maximum dans la liste fournie en paramètre, tandis que Sort trie une liste de couples $\langle P_i, l_i \rangle$ par ordre décroissant de l_i, c'est à dire que $\langle P_j, l_j \rangle > \langle P_k, l_k \rangle$ si et seulement si $l_j > l_k$. La fonction locale $\mathcal{F}_{TS}$ qui paramètre l'algorithme 1 est :

$$\mathcal{F}_{TS} = \begin{cases} \{\langle P_i, 0 \rangle\} & \text{si } \Gamma^{-1}(i) = \emptyset \\ \mathsf{Sort}\left(\begin{array}{l} \left\{ \left\langle P_i, 1 + \mathsf{Max}_{\{\langle P_j, l_j \rangle \in M_i[k] \wedge k \in \{1..\delta^-(i)\}\}} \{l_j\} \right\rangle \right\} \\ \cup \bigcup_{k \in \{1..\delta^-(i)\}} M_i[k] \end{array} \right) & \text{sinon} \end{cases}$$

Dans une configuration légitime, le registre $M_i[\delta^-(i) + 1]$ de chaque processeur P_i contient une entrée pour chacun des ascendants de P_i et une entrée pour P_i lui-même. De plus, chaque l_j dans un couple $\langle P_j, l_j \rangle$ d'un registre $M_i[\delta^-(i) + 1]$ est égal au niveau du processeur P_j. Une feuille P_i (pour laquelle $\Gamma^{-1}(i) = \emptyset$) place continuellement $\langle P_i, 0 \rangle$ dans $M_i[\delta^-(i) + 1]$. De plus, une feuille ne peut exécuter la règle $R_1 | \mathcal{F}_{TS}$ puisqu'elle ne possède pas d'ascendant direct. La figure 4.3 présente une configuration légitime du système.

Correction de la fonction $\mathcal{F}_{TS}$

En vertu du théorème 1, le système est auto-stabilisant, c'est à dire qu'il converge vers une configuration où toutes les entrées de la fonction $\mathcal{F}_{TS}$ sont correctes sur chacun des

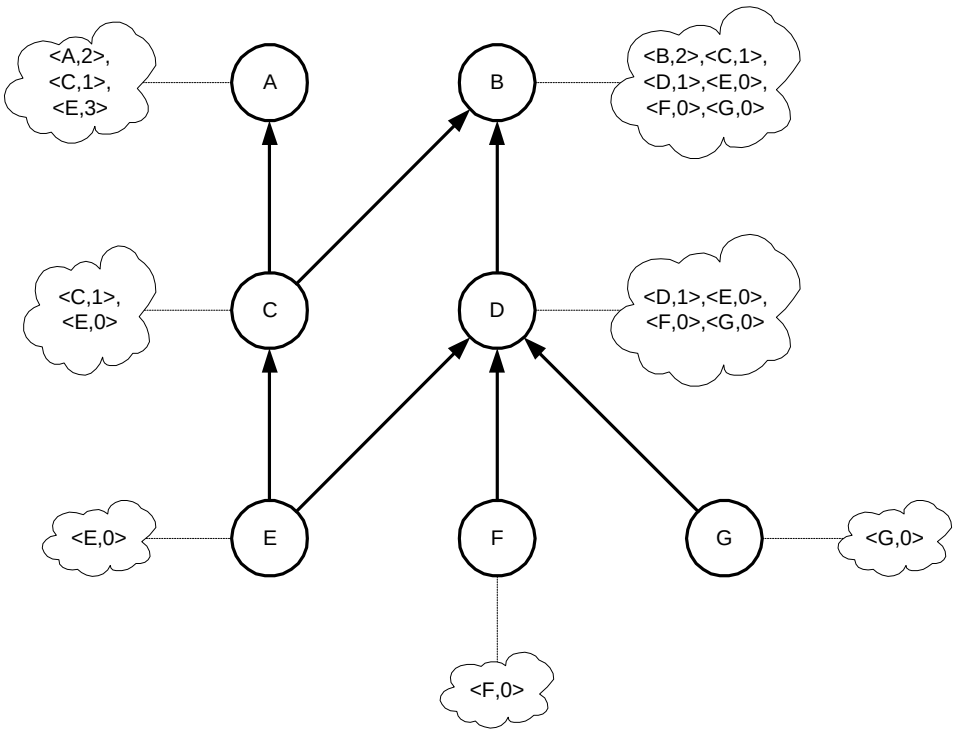

FIG. 4.3 – Un tri topologique effectué avec la fonction $\mathcal{F}_{TS}$.

processeurs. Montrons que la fonction choisie retourne des résultats corrects quand ses entrées sont correctes.

Lemme 4.17 *La fonction $\mathcal{F}_{TS}$ résout la tâche du Tri Topologique.*

Preuve: Nous devons montrer que la fonction $\mathcal{F}_{TS}$ crée des listes qui satisfont les conditions TS_1 et TS_2. Montrons tout d'abord que pour chaque processeur P_i du système, le registre $M_i[\delta^-(i)+1]$ contient au moins un couple pour chaque ascendant P_j de P_i (y compris P_i) tels que chaque couple contient l'identifiant P_j de l'ascendant, ainsi que son niveau. Prouvons par récurrence sur le niveau que c'est bien le cas.

Dans le cas d'une feuille, $\mathcal{F}_{TS}$ retourne $\langle P_i, 0\rangle$. Ceci est correct puisque les feuilles n'ont pas d'ascendants et sont toujours de niveau 0.

Si le processeur P_i n'est pas une feuille, et en supposant que tous ses ascendants directs sont corrects, l'entrée de la fonction $\mathcal{F}_{TS}$ sur P_i est l'union des sorties de tous les ascendants directs de P_i. Donc l'entrée comprend au moins un couple correct pour chacun des ascendants de P_i, excepté P_i lui-même. La fonction $\mathcal{F}_{TS}$ crée un nouveau couple $\langle P_i, l_i\rangle$ tel que l_i vaut un de plus que le maximum des niveaux relevés sur les ascendants directs de P_i. Par définition l_i est égal au niveau de P_i. Prouvons maintenant que $M_i[\delta^-(i)+1]$ est trié topologiquement, c'est à dire que chaque processeur apparaît avant tous ses ascendants. Puisque $\mathcal{F}_{TS}$ effectue un tri de $M_i[\delta^-(i)+1]$ par ordre décroissant de niveau, le processeur dont le niveau est le plus élevé est placé en premier. Puisque chaque processeur qui n'est pas une feuille a un niveau qui est supérieur au moins de 1 au niveau de chacun de ses ascendants directs (et *a fortiori* de ses ascendants), la liste stockée dans le registre $M_i[\delta^-(i)+1]$ est triée topologiquement. □

4.5.2 Calcul de Distance, Arbre et Forêt de Longueur Minimale

Calculer la distance à un nœud principal du graphe de communication nécessite le calcul des longueurs des chemins dont l'origine est u. Un tel calcul peut se faire en utilisant l'incrémentation ($x \mapsto x+1$) sur les arcs. De plus, si deux chemins différents partant de u arrivent au nœud v par deux arcs incidents différents, alors v devrait choisir le chemin dont la longueur est la plus petite, c'est à dire que le processeur P_v devrait effectuer un calcul de minimum. Intuitivement, un opérateur tel que $(x, y) \mapsto \mathsf{min}(x, y+1)$ devrait résoudre le problème. Prouvons que cet opérateur est correct et supposons que les registres du système réparti sont larges de k bits, et que 2^k est supérieur à toute distance de u à n'importe quel autre nœud du réseau.

Lemme 4.18 *L'opérateur* minc *est un r-opérateur strictement idempotent sur l'ensemble $\mathbb{S} = \{0, \ldots, 2^k - 1\}$.*

Preuve: Soit $\mathbb{S}$ l'ensemble *fini* $\{0, \ldots, 2^k - 1\}$ (hypothèse 4.2). L'opérateur min est un s-opérateur sur $\mathbb{S}$ et définit un ordre *total* dénoté par $<$ (hypothèse 4.1). Son élément neutre est $2^k - 1$. Soit r une fonction de $\mathbb{S}$ vers $\mathbb{S}$ définie par $\mathsf{r}(x) \equiv x + 1$ pour $0 \leq x < 2^k - 1$

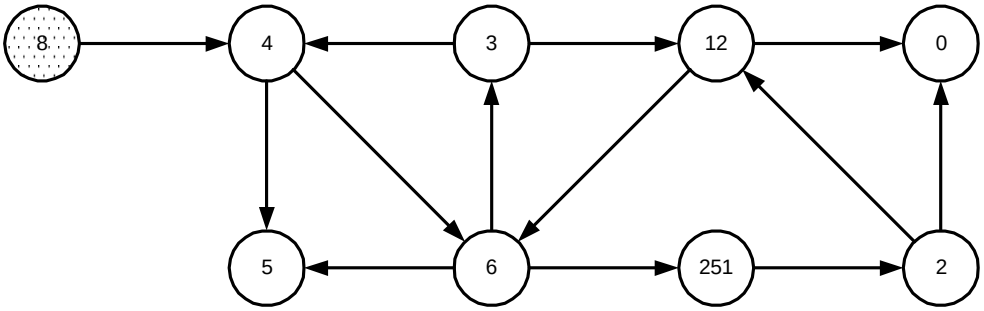

FIG. 4.4 – Une configuration incorrecte du système instancié par minc.

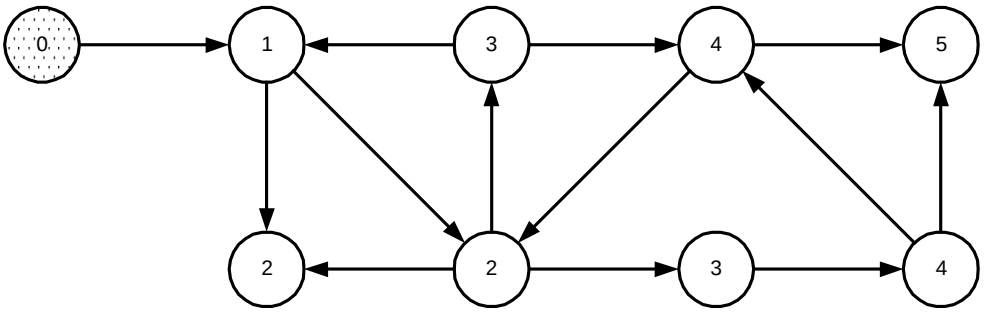

FIG. 4.5 – Une configuration après stabilisation du système instancié par minc.

et $\mathsf{r}(2^k-1) \equiv 2^k-1$. Cette fonction est un homomorphisme de $(\mathbb{S}, \mathsf{min})$. Soit $\mathsf{minc}(x,y)$ le r-opérateur binaire défini sur $\mathbb{S}$ par $\mathsf{minc}(x,y) = \mathsf{min}(x, \mathsf{r}(y))$ (définition 4.3). Puisque $x < \mathsf{r}(x) = x+1$ pour tout $x \in \mathbb{S} \setminus \{2^k-1\}$, minc est un r-opérateur strictement idempotent sur $\mathbb{S}$ (définition 4.4). □

Soit $\mathcal{S}$ un système réparti tel que le graphe de communication admette un nœud principal dont la donnée initiale $M_u[0] = 0$ et dont la donnée initiale de chacun des autres nœuds $v \neq u$ soit $M_v[0] = 2^k-1$. Une telle hypothèse revient à distinguer un nœud principal sur le réseau. D'après la proposition 4.7, l'algorithme 1 est auto-stabilisant quand il est instancié avec l'opérateur minc comme $\mathcal{F}_v$. Quand le système est stabilisé, chaque nœud w possède la valeur légitime suivante, où $\mathsf{r}_{P_{v\to w}}$ dénote la composition des r-fonctions le long du chemin $P_{v\to w}$:

$$M[\delta^-(w)+1] = \mathsf{min}\left\{\mathsf{r}_{P_{v\to w}}(M_v[0]), v \in \Gamma^-_{\mathcal{G}}(w), P_{v\to w} \text{ un chemin élémentaire de } v \text{ à } w\right\}$$

ce qui est équivalent à $M[\delta^-(w)+1] = \mathsf{r}^{Dist(u,w)}(M_u[0]) = Dist(u,w)$. Par conséquent, le r-opérateur minc calcule la distance à un nœud principal sur tout réseau de manière auto-stabilisante.

Si chaque nœud dispose d'un moyen lui permettant d'identifier l'indice de l'arc incident choisi dans le calcul du minimum, un arbre couvrant des chemins de longueur minimale se

déduit à partir des arcs choisis. La figure 4.4 montre une configuration incorrecte due à une défaillance transitoire, tandis que la figure 4.5 montre une configuration légitime obtenue après stabilisation de l'algorithme. S'il existe plusieurs nœuds principaux différenciés dans le système, chaque nœud va se retrouver dans l'arbre dont la racine est la plus proche de lui. Ceci résout le problème de la forêt couvrante de longueur minimale.

4.5.3 Arbre et Forêt de plus Courts Chemins

Le problème du calcul du plus court chemin à un nœud principal est similaire à celui du calcul de la distance, si ce n'est que les poids associés à chaque arc ne sont plus nécessairement égaux à 1. Notons ω_v^i le poids du $i^{\text{ième}}$ arc incident du nœud v. Montrons que l'opérateur minc_ω défini par :

$$\mathsf{minc}_\omega\left(x_0,\ldots,x_{\delta^-(v)}\right) \equiv \mathsf{min}\left(2^{k-1}, x_0, x_1+\omega_v^1, \ldots, x_{n-1}+\omega_v^{\delta^-(v)}\right)$$

est un r-opérateur strictement idempotent. Comme lors de la section 4.5.2, nous considérons le même ensemble *fini* $\mathbb{S}$, en supposant que 2^k-1 est supérieur à toute distance pondérée dont l'origine est u.

Lemme 4.19 *L'opérateur* minc_ω *est un* r*-opérateur strictement idempotent sur l'ensemble* $\mathbb{S}=\{1,\ldots,2^k-1\}$.

Preuve: L'opérateur $\oplus$ défini par $x\oplus y=\mathsf{min}(2^k-1,x,y)$ est un s-opérateur sur $\mathbb{S}$ qui définit un *ordre total* sur $\mathbb{S}$. Par suite, les hypothèses 4.1 et 4.2 sont vérifiées. Pour chaque poids ω_v^i, la fonction r_v^i définie par $x\mapsto \mathsf{min}(2^k-1,x+\omega_v^i)$ est un homomorphisme de $(\mathbb{S},\oplus)$: $\mathsf{r}(x\oplus y)=\mathsf{min}(2^k-1,\mathsf{min}(2^k-1,x,y)+\omega_v^i)=\mathsf{min}\left(2^k-1,\mathsf{min}(2^k-1,x+\omega_v^i),\mathsf{min}(2^k-1,y+\omega_v^i)\right)=\mathsf{r}_v^i(x)\oplus\mathsf{r}_v^i(y)$.

Nous définissons alors le r-opérateur n-aire minc_ω sur $\mathbb{S}=\{0,1,\ldots,2^k-1\}$ comme suit :

$$\begin{aligned}\mathsf{minc}_\omega\left(x_0,\ldots,x_{\delta^-(v)}\right) &= \mathsf{min}\left(x_0,\mathsf{min}(2^{k-1},x_1+\omega_v^1),\ldots,\mathsf{min}(2^{k-1},x_{n-1}+\omega_v^{\delta^-(v)})\right)\\ &= \mathsf{min}\left(2^{k-1},x_0,x_1+\omega_v^1,\ldots,x_{n-1}+\omega_v^{\delta^-(v)}\right)\end{aligned}$$

Si chaque ω_v^i est strictement supérieur à 0, alors chaque r-fonction $\mathsf{r}_v^i(x)=\mathsf{min}(2^k-1,x+\omega_v^i)$ vérifie $x\prec_\oplus \mathsf{r}_v^i(x)$ pour tout $x\in\mathbb{S}\setminus\{2^k-1\}$. Le r-opérateur n-aire minc_ω est donc strictement idempotent (définition 4.6). □

Comme dans l'application précédente, on suppose que le nœud principal u a sa donnée initiale $M_u[0]=0$ et que les autres nœuds $v\neq u$ ont leur donnée initiale $M_v[0]=2^k-1$. D'après la proposition 4.7, l'algorithme 1 est auto-stabilisant s'il est instancié avec l'opérateur minc_ω. Une fois le système stabilisé, chaque nœud w possède la valeur légitime suivante :

$$M[\delta^-(w)+1]=\mathsf{min}\left\{\mathsf{r}_{P_{v\to w}}(M_v[0]), v\in\Gamma_{\mathcal{G}}^-(w), P_{v\to w}\text{ un chemin élémentaire de }v\text{ à }w\right\}$$

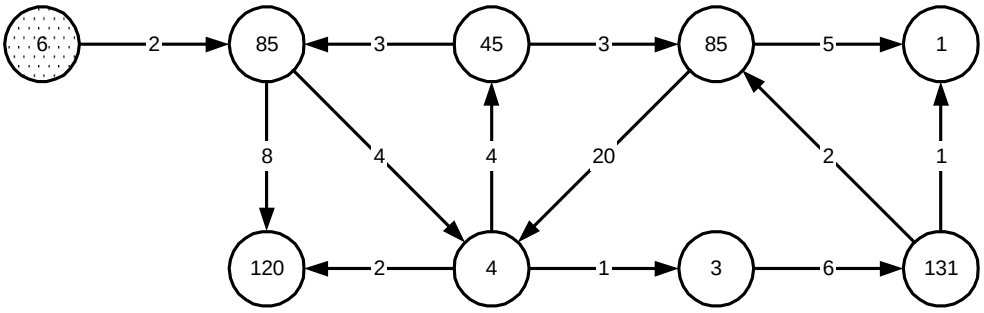

FIG. 4.6 – Une configuration incorrecte du système instancié par minc_ω.

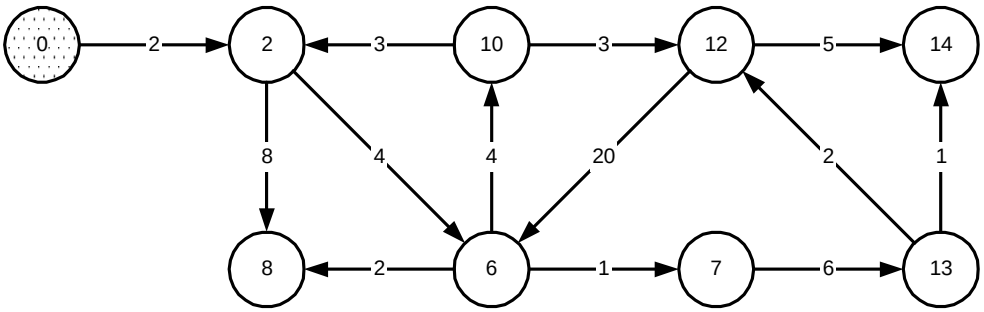

FIG. 4.7 – Une configuration après stabilisation du système instancié par minc_ω.

ce qui est équivalent à $M[\delta^-(w)+1] = \mathsf{r}_{P_{u\to w}}(M_u[0])$, où $\mathsf{r}_{P_{u\to w}}(M_u[0])$ est la plus petite distance pondérée de u à w. Par conséquent, le r-opérateur minc_ω calcule la distance pondérée à un nœud principal dans tout réseau et de manière auto-stabilisante.

Si chaque nœud dispose d'un moyen lui permettant d'identifier l'indice de l'arc incident choisi dans le calcul du minimum, un arbre couvrant des plus courts chemins se déduit à partir des arcs choisis. La figure 4.6 montre une configuration incorrecte due à une défaillance transitoire, tandis que la figure 4.7 montre une configuration légitime obtenue après stabilisation de l'algorithme. S'il existe plusieurs nœuds principaux différenciés dans le système, chaque nœud se retrouvera dans l'arbre dont la racine est la plus proche de lui. Ceci résout le problème de la forêt couvrante des plus courts chemins.

4.5.4 Arbre et Forêt des plus Fiables Émetteurs

Dans les réseaux de communications soumis à des défaillances et où les processeurs doivent choisir le meilleur émetteur possible, la distance n'est pas toujours le critère adapté. Quand le taux de défaillances des connexions avec les voisins est calculable et qu'il reste plus ou moins constant, il est intéressant de déterminer le retransmetteur pour lequel le taux de défaillances est le plus bas, et de connaître le chemin vers un tel retransmetteur.

Supposons que tous les registres soient larges de k bits et que les taux de fiabilité des lignes soient échantillonnées de 0 (ligne hors-service) à 2^k-1 (ligne parfaitement fiable). Le taux de fiabilité d'un chemin P constitué de p arcs (v_i, v_{i+1}) dont le taux de fiabilité est τ_i est obtenu par : $\tau_P = \lfloor(\Pi_{i=1}^{p}\tau_i)/(2^k-1)^{p-1}\rfloor$. Par conséquent, pour connaître le chemin le plus fiable depuis un émetteur, chaque nœud doit effectuer des multiplications et des calculs de maximum pour distinguer les arcs incidents dont le taux de fiabilité est le plus élevé. Construisons maintenant un r-opérateur qui effectue ce calcul et qui doit permettre de résoudre le problème de manière auto-stabilisante. Soit τ_v^i le taux de fiabilité du $i^{\text{ème}}$ arc incident du nœud v ; montrons que l'opérateur maxmul_τ défini par

$$\mathsf{maxmul}_\tau(x_0,\ldots,x_{n-1}) = \mathsf{max}\left(x_0, \lfloor x_1\times\tau_v^1/(2^k-1)\rfloor, \ldots, \lfloor x_{n-1}\times\tau_v^{n-1}/(2^k-1)\rfloor\right)$$

est un r-opérateur strictement idempotent.

Lemme 4.20 *L'opérateur* maxmul_τ *est un* r*-opérateur strictement idempotent sur l'ensemble* $\mathbb{S} = \{1,\ldots,2^k-1\}$.

Preuve: Soit $\mathbb{S}$ l'ensemble *fini* $\{0,\ldots,2^k-1\}$ (hypothèse 4.2). L'opérateur max est un s-opérateur sur $\mathbb{S}$ qui définit une relation d'ordre *total* $\preceq_{\mathsf{max}}$ qui est, en fait, la relation d'ordre usuelle $\geq$ sur les entiers (hypothèse 4.1). Nous supposons en outre qu'il n'existe aucun arc dont le taux de fiabilité est égal à 2^k-1 (sur chaque lien, il y a de temps en temps des défaillances). Alors, $\tau_v^i \in \mathbb{S}\setminus\{2^k-1\}$. Pour chaque taux τ_v^i, la fonction r_v^i définie par $\mathsf{r}_v^i(x)\mapsto\lfloor x\times\tau_v^i/(2^k-1)\rfloor$ est un homomorphisme de $(\mathbb{S},\mathsf{max})$. En effet,

$$\begin{aligned}
\mathsf{r}_v^i(\mathsf{max}(x,y)) &= \lfloor\mathsf{max}(x,y)\times\tau_v^i/(2^k-1)\rfloor\\
&= \lfloor\mathsf{max}(x\times\tau_v^i/(2^k-1), y\times\tau_v^i/(2^k-1))\rfloor\\
&= \mathsf{max}(\lfloor x\times\tau_v^i/(2^k-1)\rfloor, \lfloor y\times\tau_v^i/(2^k-1)\rfloor)\\
&= \mathsf{max}(\mathsf{r}_v^i(x), \mathsf{r}_v^i(y))
\end{aligned}$$

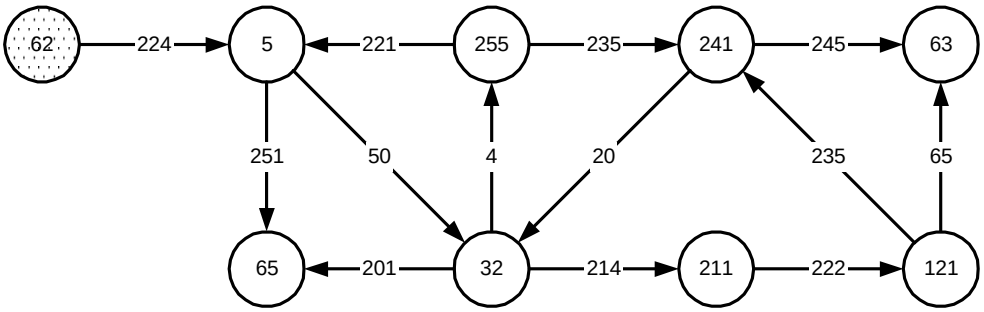

FIG. 4.8 – Une configuration incorrecte du système instancié par maxmul_τ.

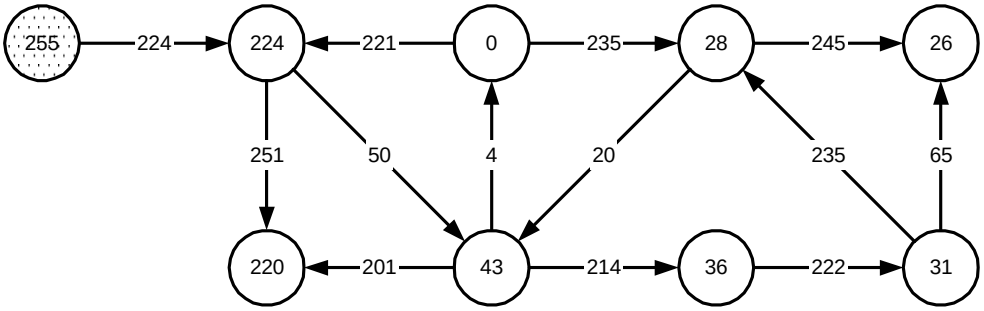

FIG. 4.9 – Une configuration après stabilisation du système instancié par maxmul_τ.

Nous définissons le r-opérateur n-aire maxmul_τ sur $\mathbb{S}$ comme suit :

$$\mathsf{maxmul}_\tau(x_0,\ldots,x_{n-1}) = \mathsf{max}\left(x_0, \lfloor x_1 \times \tau_v^1/(2^k-1)\rfloor, \ldots, \lfloor x_{n-1} \times \tau_v^{n-1}/(2^k-1)\rfloor\right)$$

Puisque $0 \leq \tau_v^i < 2^k - 1$, nous avons $x \prec_{\mathsf{max}} \mathsf{r}_v^i(x)$ pour chacune des r-fonctions r_v^i (ce qui signifie que $x > \mathsf{r}_v^i(x)$). Donc le r-opérateur n-aire est strictement idempotent. □

Supposons que pour chaque émetteur u, $M_u[0] = 2^k - 1$ et que pour tous les autres nœuds v, $M_v[0] = 0$. D'après la proposition 4.7, l'algorithme 1 est auto-stabilisant s'il est instancié avec l'opérateur maxmul_τ. Une fois le système stabilisé, chaque nœud w possède la valeur légitime suivante :

$$M[\delta^-(w)+1] = \mathsf{max}\left\{\mathsf{r}_{P_{v\to w}}(M_v[0]), v \in \Gamma_{\mathcal{G}}^-(w), P_{v\to w} \text{ un chemin élémentaire de } v \text{ à } w\right\}$$

ce qui est équivalent à $M_w[\delta^-(w)+1] = \mathsf{r}_{P_{u\to w}}(M_u[0])$, où $\mathsf{r}_{P_{u\to w}}(M_u[0])$ est le chemin de fiabilité maximale d'un ascendant émetteur u au nœud w. Par conséquent, le r-opérateur maxmul_ω calcule le taux de fiabilité maximale sur chaque nœud.

Si chaque nœud dispose d'un moyen lui permettant d'identifier l'indice de l'arc incident choisi dans le calcul du minimum, un arbre couvrant des chemins de fiabilité maximale se déduit à partir des arcs choisis. La figure 4.8 montre une configuration incorrecte due à une

défaillance transitoire, tandis que la figure 4.9 montre une configuration légitime obtenue après stabilisation de l'algorithme. S'il existe plusieurs nœuds principaux différenciés dans le système, chaque nœud se retrouvera dans l'arbre dont la racine est la plus proche de lui. Ceci résout le problème de la forêt couvrante de chemins de fiabilité maximale. Notons que notre approche rend l'algorithme *adaptatif aux entrées* : quand les taux de fiabilité sont modifiés pendant l'exécution répartie de l'algorithme, alors la forêt des chemins de fiabilité maximale est mise à jour sans requérir d'intervention extérieure.

4.5.5 Arbre en Profondeur

Dans cette section, nous présentons un r-opérateur qui permet de construire un arbre en profondeur enraciné sur un nœud principal u du graphe de communication, en dépit de défaillances transitoires.

Pour résoudre ce problème fondamental, nous supposons que chaque processeur possède un identifiant constant et unique sur le réseau, et qu'un ordre total peut être défini sur ces identifiants. Considérons une liste ordonnée d'identifiants de processeurs ; $\emptyset$ dénote la liste vide. Soit min l'opérateur binaire qui renvoie la liste la plus petite selon l'ordre lexicographique. Par exemple, si les identifiants sont les lettres $a, b, c, \ldots$ ordonnées par l'ordre alphabétique, l'opérateur min donne les résultats suivants $\mathsf{min}\,((a,b,d,e,),(a,b,c,d,e,f)) = (a,b,c,d,e,f)$, $\mathsf{min}\,((a,a),(a)) = (a)$.

Soit L la plus grande liste (au sens de min) qui peut être stockée dans les registres de k-bits de large du système réparti $\mathcal{S}$. Nous supposons en outre qu'il n'existe aucun chemin élémentaire P dans le réseau qui soit tel que la liste composée des identifiants des nœuds de P soit supérieure à L (ce qui signifie que les registres sont de taille suffisante pour stocker toutes les listes nécessaires). Montrons maintenant que l'opérateur $\mathsf{lexicat}$ défini par

$$\mathsf{lexicat}(l_1, l_2) = \mathsf{min}(l_1, \mathsf{min}(L, l_2 \cup \{v\}))$$

où v est l'identifiant du processeur qui exécute l'opérateur $\mathsf{lexicat}$, est un r-opérateur.

Lemme 4.21 *L'opérateur* $\mathsf{lexicat}$ *est un r-opérateur strictement idempotent sur l'ensemble* $\mathbb{S} = \{\emptyset, \ldots, L\}$.

Preuve: Considérons l'ensemble $\mathbb{S}$ composé des liste d'identifiants l vérifiant $\mathsf{min}(l, L) = l$. Puisque $\mathbb{S}$ est l'ensemble *fini* $\{\emptyset, \ldots, L\}$ (hypothèse 4.2), et l'opérateur $\oplus$ défini par $l_1 \oplus l_2 = \mathsf{min}(L, l_1, l_2)$ est un s-opérateur sur $\mathbb{S}$ qui définit un ordre *total* (hypothèse 4.1). Son élément neutre est L.

Pour chaque nœud v, considérons les r-fonctions r_v définies sur $\mathbb{S}$ par $\mathsf{r}_v(l) = \mathsf{min}(L, l \cup v)$, où v est l'identifiant du nœud et où $\cup$ dénote l'opérateur de concaténation de deux listes. De telles fonctions sont des homomorphismes de $(\mathbb{S}, \oplus)$: $\mathsf{r}_v(l_1 \oplus l_2) = \mathsf{min}(L, \mathsf{min}(L, l_1, l_2) \cup v) = \mathsf{min}(L, \mathsf{min}(L, l_1 \cup v), \mathsf{min}(L, l_2 \cup v)) = \mathsf{r}_v(l_1) \oplus \mathsf{r}_v(l_2)$.

Définissons ensuite le r-opérateur binaire $\mathsf{lexicat}$ sur $\mathbb{S}$ par $\mathsf{lexicat}(l_1, l_2) = l_1 \oplus r(l_2)$. Pour chaque liste $l \in \mathbb{S} \setminus \{L\}$, $l \prec_{\oplus} \mathsf{r}(l)$. Le r-opérateur $\mathsf{lexicat}$ est donc strictement idempotent. □

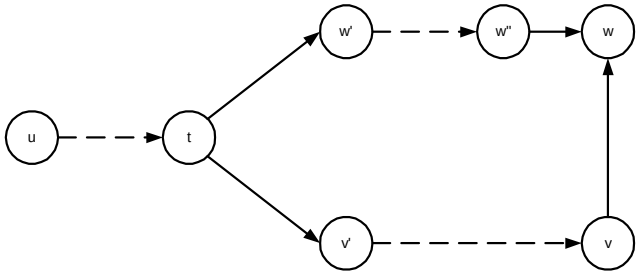

FIG. 4.10 – Construction d'un arbre en profondeur.

Supposons que le nœud principal u (racine de l'arbre en profondeur) soit tel que $M_u[0] = \emptyset$ tandis que les autres nœuds $v \neq u$ sont tels que $M_v[0] = L$. D'après la proposition 4.7, l'algorithme 1 est auto-stabilisant quand il est instancié par l'opérateur lexicat pour $\mathcal{F}_v$. Une fois le système stabilisé, chaque nœud w possède la valeur légitime suivante :

$$M[\delta^-(w)+1] = \mathsf{min}\left\{\mathsf{r}_{P_{v\to w}}(M_v[0]), v \in \Gamma^-_{\mathcal{G}}(w), P_{v\to w} \text{ un chemin élémentaire de } v \text{ à } w\right\}$$

ce qui est équivalent à $M[\delta^-(w)+1] = \mathsf{r}_{P_{u\to w}}(M_u[0])$, où $\mathsf{r}_{P_{u\to w}}(M_u[0])$ est la plus petite (au sens de min) liste d'identifiants de tous les chemins du nœud u au nœud v.

Prouvons maintenant que ce résultat constitue un arbre en profondeur enraciné sur u.

Premièrement, toutes les valeurs légitimes sont des listes commençant par u, l'identifiant de la racine. De plus, tous les nœuds $v \neq u$ sont tels que $M_v[1]..M_v[\delta^-(v)]$ contient une liste plus petite que leur propre valeur légitime $M_v[\delta^-(v)+1]$. Cette liste la plus petite est la valeur légitime d'un ascendant direct de v. En choisissant l'arc incident correspondant, on obtient un arbre enraciné en u, que nous dénotons T_u.

Pour prouver que cet arbre est bien un arbre en profondeur, il suffit de vérifier que la numérotation de chaque nœud donnée par la valeur légitime est une numérotation en profondeur d'abord du graphe de communication $\mathcal{G}$:

$$\forall v, w \in V, \quad w \in \Gamma^-_{\mathcal{G}}(v) \Rightarrow \left\{ \begin{array}{ll} & w \in \Gamma^-_{T_u}(v) \quad \text{et} \quad M_v[\delta^-(v)+1] \prec_{\oplus} M_w[\delta^-(w)+1] \\ \text{ou} & \\ & w \notin \Gamma^-_{T_u}(v) \quad \text{et} \quad M_w[\delta^-(w)+1] \prec_{\oplus} M_v[\delta^-(v)+1] \end{array} \right.$$

Les valeurs légitimes sont toutes uniques puisqu'elles se rapportent à un chemin unique. De plus, le numéro croît le long d'un chemin de T_u. Par conséquent, si cette numérotation n'est pas une numérotation en profondeur, c'est qu'il existe un arc $(v, w) \in \mathcal{G}$ tel que $M_v[\delta^-(v)+1] \prec_{\oplus} M_w[\delta^-(w)+1]$ et $w \notin T_v$ où T_v dénote le sous-arbre de T_u enraciné en v. Dans ce cas, nous avons $w \notin T_v$ et $v \notin T_w$ (voir figure 4.10).

Alors il existe un nœud t qui est un ascendant commun à v et w dans T_u et néanmoins différent à la fois de v et de w. Soit v' le premier nœud sur le chemin de t à v dans T_u et soit w' (respectivement w'') le premier (respectivement le dernier) nœud sur le chemin de

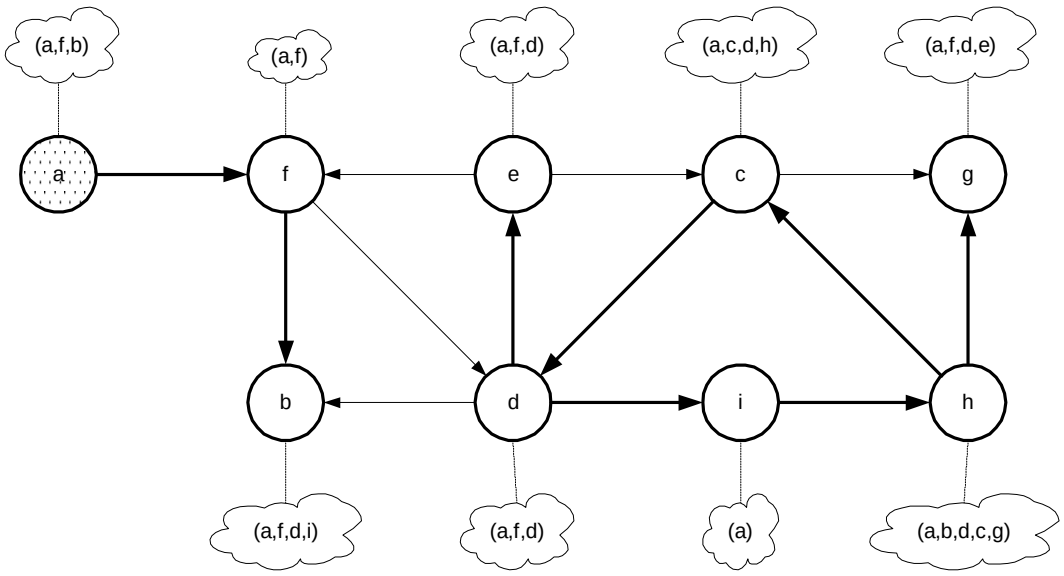

FIG. 4.11 – Une configuration incorrecte du système instancié par lexicat.

t à w. Nous avons $M_v[\delta^-(v)+1] = M_t[\delta^-(t)+1] \cup (v') \cup \cdots \cup (v)$ et $M_w[\delta^-(w)+1] = M_t[\delta^-(t)+1] \cup (w') \cup \cdots \cup (w'') \cup (w)$. Puisque $M_v[\delta^-(v)+1] \prec_\oplus M_w[\delta^-(w)+1]$, nous avons $(v') \prec_\oplus (w')$. Par suite, la liste incidente reçue par w de son ascendant direct v $(M_t[\delta^-(t)+1] \cup (v') \cup \cdots \cup (v))$ est plus petite (au sens de l'ordre lexicographique induit par $\oplus$) que celle reçue de son ancêtre direct w'' $(M_t[\delta^-(t)+1] \cup (w') \cup \cdots \cup (w''))$, ce qui contredit le fait que w avait choisi l'un de ses ascendants directs pour construire l'arbre en profondeur T_u.

En définitive, T_u est un arbre en profondeur enraciné sur le nœud principal u ; les valeurs légitimes donnent une numérotation en profondeur, et contiennent la liste des nœuds de la racine à eux-mêmes dans l'arbre. La distance à la racine s'en trouve trivialement calculée (longueur de la liste).

La figure 4.11 montre une configuration incorrecte due à une défaillance transitoire, tandis que la figure 4.12 montre une configuration légitime obtenue après stabilisation de l'algorithme.

4.6 Cas des Ensembles Infinis

Dans cette section, nous nous intéressons à la validité des résultats précédents dans l'hypothèse où l'ensemble des données est infini. Notons tout d'abord que dans le cas d'un graphe sans cycle, aucune hypothèse n'a été faite concernant le caractère fini de l'ensemble de définition de la fonction utilisées en paramètre de l'algorithme. Les résultats montrés pour les graphes sans cycles demeurent valides sans modifications.

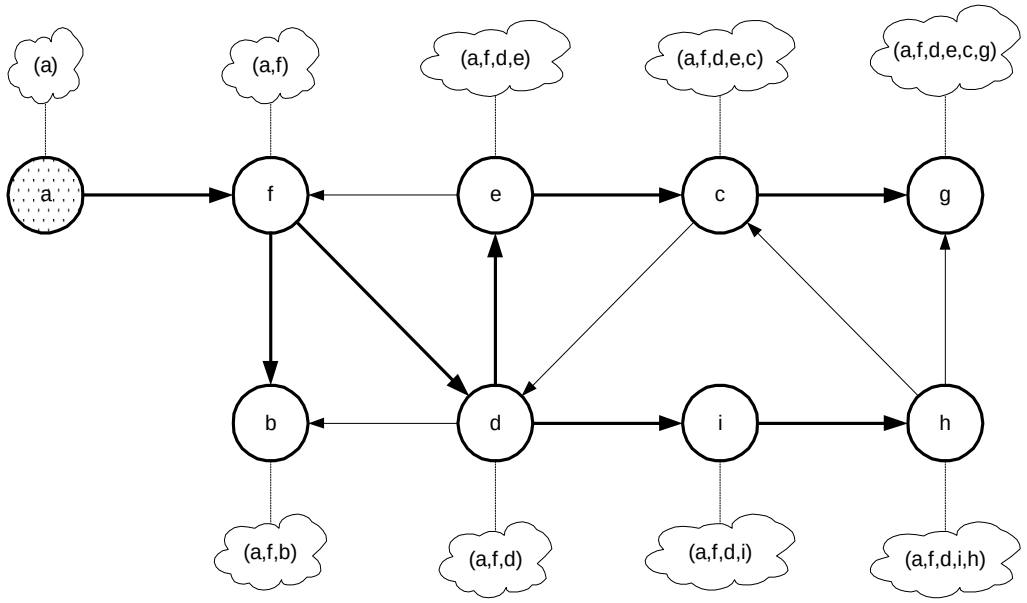

FIG. 4.12 – Une configuration après stabilisation du système instancié par lexicat.

Dans le cas général, l'hypothèse 4.2 que nous avons faite contraint l'ensemble de définition du r-opérateur utilisé à être fini. Or, seul le lemme 4.5 fait usage de cette hypothèse. Dans cette section, nous donnons l'intuition de la preuve que le lemme est toujours valide dans le cas où l'ensemble de définition est infini, mais est tel que toute suite strictement croissante est non bornée. Cette propriété est vérifiée par les ensembles $\mathbb{N}$ et $\mathbb{Z}$, mais ne l'est pas par les ensembles $\mathbb{Q}$ ou encore $\mathbb{R}$. Nous supposons de plus que tout processeur possède parmi ses ascendants un processeur dont la donnée initiale est différente de $e_{\oplus}$. Enfin, la valeur $e_{\oplus}$ n'est pas admise comme valeur valide dans une variable $M_i[k]$ où $k \in \{1, \ldots, \delta^-(i) + 1\}$. Dans le cas où elle serait présente suite à une défaillance transitoire, un processeur est autorisé à la remplacer par une autre valeur arbitraire.

La preuve du lemme 4.5 est basée sur le fait qu'une valeur ne peut se déplacer indéfiniment sur un cycle. En supposant l'ensemble infini, et en supposant qu'il existe un cycle tel qu'une valeur croisse indéfiniment sur celui-ci, alors il existe au moins un processeur (soit sur le cycle, soit ailleurs dans le réseau) dont la donnée initiale est différente de $e_{\oplus}$ (par hypothèse). Si ce processeur est sur le cycle, alors il existe un moment ou la valeur croissante dépasse sa donnée initiale, ce qui l'empêche de se propager de nouveau. Si ce processeur n'est pas sur le cycle, il est atteignable par un processeur P_i du cycle, au moyen de la variable locale $M_i[k]$, pour un $k \in \{1, \ldots, \delta^-(i) + 1\}$. Comme cette valeur est différente de $e_{\oplus}$, la valeur croissante dépasse la variable $M_i[k]$, ce qui l'empêche de se propager à nouveau. Dans tous les cas, une propagation indéfinie d'une variable est impossible.

Le problème qui apparaît au vu des hypothèses supplémentaires nécessaires pour assurer la stabilisation du système, est que l'on passe d'une condition purement locale (le paramètre

de l'algorithme 1 est un r-opérateur strictement idempotent) à une condition partiellement globale (tout processeur possède un ascendant dont la donnée initiale est différente de $e_{\oplus}$). Un moyen de repasser à une condition purement locale est d'imposer que tous les processeurs ne puissent jamais avoir $e_{\oplus}$ comme donnée initiale.

4.7 Résumé

Dans ce chapitre, nous avons montré qu'une condition locale facilement vérifiable car portant sur l'algorithme exécuté localement par un processeur était suffisante pour garantir l'auto-stabilisation du système tout entier. La propriété d'auto-stabilisation ainsi obtenue est indépendante de la topologie du graphe de communication sous-jacent, de la politique d'ordonnancement des processeurs, et de la granularité de l'atomicité des programmes. De surcroît, notre approche n'utilise pas de connaissances globales particulières comme le nombre de processeurs présents ou une borne sur le diamètre du graphe de communication.

Nous présentons plusieurs opérateurs qui vérifient cette condition locale, et qui apportent une solution à plusieurs problèmes statiques fondamentaux. Utiliser de tels opérateurs présente deux intérêts majeurs : aucune preuve n'est nécessaire autre que la satisfaction locale de la condition, et une implantation efficace de ces opérateurs existe pour des systèmes réels (voir [Duc99]).

Chapitre 5

Synchronisation

Le chapitre 3 a montré qu'il était impossible de construire — en toute généralité — un mécanisme utilisant une mémoire finie et détectant le mauvais fonctionnement d'une tâche dynamique équitable. Il est donc tout particulièrement intéressant d'obtenir des solutions auto-stabilisantes à ce genre de tâches. Dans ce chapitre, nous considérons la tâche de la synchronisation : garantir qu'un certain sous-ensemble des processeurs agit de manière quasi-simultanée. Bien entendu, pour que cette tâche ne soit pas trivialement résolue, nous supposons que les exécutions du système ne sont pas restreintes par le démon synchrone, mais qu'elles sont au contraire soumises au démon réparti.

5.1 Références

Du fait que les systèmes répartis sont le plus souvent constitués par des processeurs exécutant des instructions à des vitesses différentes, la tâche de synchronisation est cruciale pour aider les concepteurs de systèmes à implanter des algorithmes parallèles dans des environnements asynchrones. Informellement, un synchroniseur est un protocole qui permet de simuler le comportement de systèmes synchrones. Plusieurs solutions au problème de synchronisation ont été présentées par Awerbuch dans [Awe85], et ont fait l'objet d'une étude en profondeur dans [RH90, Tel94, Lyn96]. Dans de nombreux systèmes répartis, les vagues de communication (des phases répétées de diffusion et de concentration) sont utilisées de manière à garantir une synchronisation entre des processeurs.

Après le travail initial en auto-stabilisation de Dijkstra [Dij74], quelques schémas généraux pour rendre stabilisants les protocoles non-stabilisants d'Awerbuch ont été proposés. En particulier, la combinaison d'un synchroniseur non-stabilisant et d'un protocole de réinitialisation auto-stabilisant ont été étudiés. Le protocole de réinitialisation de [AG94] stabilise en temps $O(D^2)$, où D est une borne supérieure sur le diamètre du réseau. Dans [APSV91], une méthode générale de réinitialisation sur un réseau utilisant une détection locale et une correction globale a été introduite, qui stabilise en un temps $O(N)$, où N est le nombre de processeurs du système. De manière indépendante, le synchroniseur présenté dans [AKM$^+$93]

stabilise en un temps optimal, c'est à dire $O(d)$, mais occupe un espace $O(\log N \log D)$, où d est de diamètre véritable du réseau.

Des algorithmes auto-stabilisants de construction d'arbres ont été proposés dans [AKY90, AG94, AKM$^+$93, CYH91, DIM93, Joh97]. Chacun de ces algorithmes peut être combiné avec un synchroniseur pour des réseaux en arbres pour construire un synchroniseur pour un réseau général.

Le paradigme de la propagation de compteur introduite dans [Var94] offre un principe général pour la synchronisation globale dans les réseaux en arbre. Cette technique stabilise en temps $O(h)$ et occupe un espace $O(\log N)$ pour implanter les compteurs, où h est la hauteur de l'arbre. Un algorithme optimal en espace pour la propagation d'informations avec retour (PIF) dans les réseaux en arbres est présentée dans [BDPV98]. Le temps de stabilisation de cet algorithme est $h-1$ actions et l'espace requis est de 3 états par processeur.

La notion de configuration légitime telle qu'elle est décrite dans [Dij74] a été modifiée dans [Kru79] pour permettre des exécutions concurrentes dans des branches différentes d'un réseau en arbre. Bien que l'article de Kruijer ne discute pas du problème de synchronisation, l'algorithme présenté peut également être considéré comme un sychoniseur global dans des réseaux en arbre. D'ailleurs, l'algorithme 2 présenté dans la section 5.3.3 est semblable à celui proposé par Kruijer, mais dans la section 5.3.4 nous prouvons qu'il est auto-stabilisant même sous le démon réparti, alors que le résultat de Kruijer n'est valable que sous le démon central.

La *synchronisation locale* est un relâchement des contraintes de synchronisation globale : seulement les *voisins* d'un processeur doivent exécuter leurs actions de manière quasi-simultanée. Dans [GH97], Gouda et Haddix proposent un alternateur auto-stabilisant sur un réseau en chaîne qui transforme tout système linéaire stabilisant seulement sous le démon central en un système stabilisant qui fonctionne sous le démon réparti. Un synchroniseur local et un autre global sont présentés dans [DIM97]. Le synchroniseur local est utilisé pour synchroniser des variables dans un système constitué de deux processeurs et également un synchroniseur global dans un réseau en arbre. Le temps de stabilisation du synchroniseur global est $O(\Delta h)$ où Δ est le degré de l'arbre et h la hauteur de l'arbre.

Dans ce chapitre, nous présentons deux synchroniseurs auto-stabilisants, le premier global, le second local, pour les réseaux en arbre. Chacun d'eux occupe seulement un espace mémoire en $O(1)$ (en plus de la mémoire nécessaire à la construction de l'arbre) et stabilise en temps $O(h)$ pour le premier algorithme et en temps $O(1)$ pour le second (qui est donc *instantanément stabilisant*). Nous montrons que plusieurs algorithmes répartis peuvent être rendus facilement auto-stabilisants au moyen de ces synchroniseurs, sans que cela occasionne un surcoût rédhibitoire. En particulier, nous présentons une solution au problème de diffusion comme application du synchroniseur local. Cet algorithme de diffusion a seulement besoin de $2h+2m-1$ cycles asynchrones d'exécution pour diffuser m messages. Le synchroniseur local de [DIM97] synchronise seulement deux processeurs, alors que le synchroniseur local présenté dans ce chapitre synchronise un processeur avec chacun de ses voisins (parent et enfants dans le réseau en arbre). Pour établir un parallèle avec les travaux de [GH97], le synchroniseur

local peut être considéré comme une généralisation d'un alternateur auto-stabilisant sur des réseaux en arbre.

5.2 Hypothèses Spécifiques au Chapitre

Nous considérons que le graphe de communication est *non-orienté* et structuré en *arbre*. Le nœud racine est dénoté par r, l'ensemble des nœuds feuilles par L, et l'ensemble des nœuds intermédiaires par I. L'ensemble de tous les nœuds du graphe de communication est donc $V = \{r\} \cup I \cup L$. La hauteur de l'arbre est dénotée par h, et la hauteur du sous-arbre enraciné sur un nœud i est dénotée par h_i. Nous supposons que sur chaque processeur P_i, l'ensemble de ses enfants (dénoté par Enf_i) et son parent (dénoté par Par_i) sont disponibles de manière non volatile (ce sont des constantes).

Nous supposons que le système s'exécute sous un démon *faiblement équitable* et *réparti*. Nous utilisons la notation $Activable(A, p, \gamma)$ pour indiquer que la garde de l'action A est vraie sur le processeur p dans la configuration γ, et la notation $Activable(p, \gamma)$ si la garde d'au moins une action est vraie sur le processeur p dans la configuration γ.

5.3 Le Synchroniseur Global

Après avoir spécifié le problème à résoudre dans la section 5.3.1, nous donnons une solution informelle dans la section 5.3.2 et l'algorithme lui même dans la section 5.3.3. La preuve de correction peut être trouvée dans la section 5.3.4 et des résultats de complexité dans la section 5.3.5.

5.3.1 Spécification du Problème

De manière informelle, le problème consiste à garantir qu'entre deux actions d'un processeur du réseau, chacun des autres processeurs exécute *exactement* une action. Le fait d'exécuter une action est symbolisé par le changement de valeur d'une variable couleur sur un processeur. La couleur du processeur i est dénotée par $Couleur_i$ et peut prendre ses valeurs dans l'ensemble $\{noir, blanc\}$. Nous posons $\neg noir = blanc$ et $\neg blanc = noir$.

Nous utilisons les vagues colorées comme motif de base des exécutions abstraites de notre spécification. Plus formellement, une vague colorée est :

Définition 5.1 (Vague Colorée) *Une* vague b-colorée *est un journal restreint* $c_1, \ldots, c_k$ *tel que :*

1. *Dans la configuration* c_1*, tous les processeurs sont de la couleur* $\neg\, b$ *(Soit formellement* $\forall i \in V$, $\text{Couleur}_i = \neg\, b$*).*
2. *Dans la configuration* c_k*, tous les processeurs sont de la couleur* b *(Soit formellement* $\forall i \in V$, $\text{Couleur}_i = b$*).*

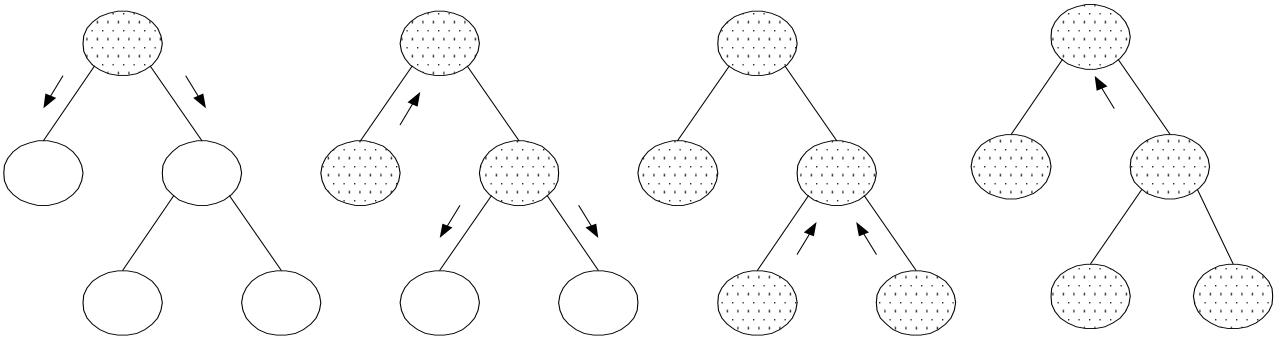

FIG. 5.1 – Vagues Colorées par Diffusion d'Informations avec Retour.

3. *Entre c_1 et c_k, tout processeur change de couleur exactement une fois (Soit formellement, $\forall i \in V$, $\exists m \geq 1$,* Couleur$_i = \neg\, b$ *dans* $c_1, \ldots, c_m$ *et* Couleur$_i = b$ *dans* $c_{m+1}, \ldots, c_k$*).*

Une vague b-colorée est notée VC^b. Le nœud racine r est l'*initiateur* de la vague colorée. Deux vagues colorées $VC_1^b = c_1, \ldots, c_k$ et $VC_2^{b'} = c'_1, \ldots c'_{k'}$ sont *enchaînables* si et seulement si $b = \neg b'$, car cela implique que $c_k = c'_1$. Leur *enchaînement* $c_1, \ldots, c_k, c'_2, \ldots c'_{k'}$ est noté $\widehat{VC_1^b VC_2^{b'}}$. La spécification de la tâche de *synchronisation globale* consiste en une séquence infinie d'enchaînements, soit formellement une séquence du type $(\widehat{VC^{noir} VC^{blanc}})^* \cup (\widehat{VC^{blanc} VC^{noir}})^*$.

5.3.2 Description Informelle

Dans les systèmes répartis habituels (c'est à dire non-stabilisants), la synchronisation par vagues colorées sur des réseaux en arbres s'effectue facilement au moyen d'algorithmes de *diffusion d'informations avec retour* (abrégé en *PIF*, de l'anglais *propagation of information with feedback*). Initialement, tous les processeurs sont de la même couleur, et la racine envoie des messages de *Diffusion* à chacun de ses fils. A leur tour, les fils de la racine propagent ces messages de diffusion à leurs propres enfants, tout en changeant de couleur. Lorsqu'un message de diffusion arrive sur une feuille de l'arbre, celle-ci change de couleur puis envoie à son parent un message de *Retour*. Ce message est alors relayé tout au long des branches de l'arbre sans pour autant occasionner de changements de couleur. La figure 5.1 illustre ce mécanisme sur un réseau en arbre de hauteur 2.

Cependant, des problèmes surviennent avec ce procédé dès que l'état du système peut être corrompu. En particulier, la figure 5.2 décrit le comportement d'un système réduit à une seule branche après des défaillances transitoires.

Plusieurs messages incorrects sont présents dans le système, et le comportement des processeurs vis à vis de ces messages fait que les exécutions du système ne satisfont plus les spécifications de la synchronisation globale : entre les deux premier changement de couleur de

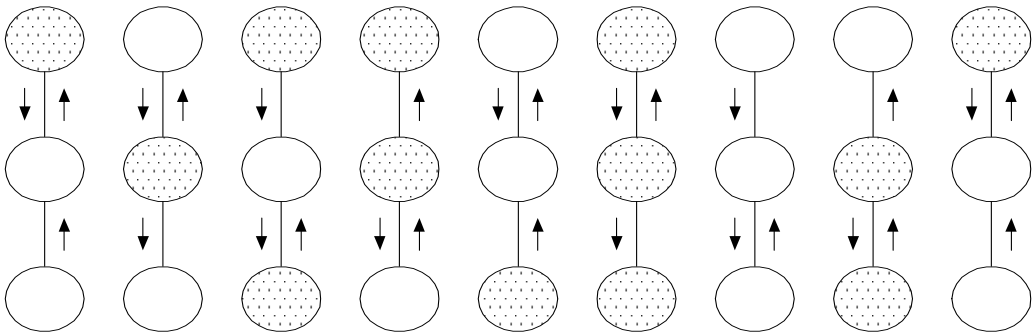

FIG. 5.2 – Un Système de Propagation d'Information avec Retour après des Défaillances Transitoires.

la racine, la feuille ne change pas de couleur ; entre les trois derniers changements de couleur de la feuille, la racine ne change de couleur que deux fois. Ce système n'est pas non plus stabilisant puisque la première et la dernière configuration sont identiques : le motif présenté peut se répéter à l'infini et le système exhibe indéfiniment un comportement incorrect.

Le synchroniseur global que nous présentons est semblable à celui de [Kru79]. Il propose simplement d'utiliser la couleur de la vague pour définir un ordre de priorité entre les phases de *Diffusion* et les phases de *Retour*. Les feuilles et les processeurs intermédiaires agissent comme s'ils étaient les esclaves de leur parent (et par suite esclaves de la racine) en copiant la couleur de leur parent tout en propageant la *Diffusion*. Seule la racine est en mesure d'utiliser une nouvelle couleur afin d'initier une nouvelle vague de communication. Fatalement, tous les processeurs finissent par avoir la même couleur que la racine, qui initie alors une nouvelle vague de communication.

1. La *racine* est considérée comme le chef d'orchestre du système. Quand sa couleur est b, elle initie une vague b-colorée en envoyant des messages de *Diffusion* b-colorés à chacun de ses enfants (qui sont soit des nœuds intermédiaires, soit des feuilles). A la réception de messages de *Retour* b-colorés de chacun de ses enfants, la racine initie une nouvelle vague colorée d'une couleur différente de b.
2. Sur réception d'un message de *Diffusion* b-colorés de leur parent, les processeurs *intermédiaires* comparent leur couleur avec celle du message. *Si* ces couleurs sont différentes, alors la couleur du parent est copiée localement, et la phase de *Diffusion* est propagée sur les enfants. Ce procédé synchronise les processeurs intermédiaires avec leur parent. Sur réception de messages de *Retour* b-colorés de chacun de leurs enfants, les processeurs intermédiaires propagent la phase de *Retour* à leur parent.
3. Les *feuilles* agissent presque de la même manière que les processeurs intermédiaires, compte tenu du fait qu'ils n'ont eux-mêmes pas d'enfants. Ils n'acceptent donc *que* les messages de *Diffusion* b-colorés de leur parent, et envoient en acquittement des messages de *Retour* b-colorés.

La différence essentielle est la *coloration* des messages. Les messages de *Diffusion* ne sont pris en compte que s'ils ont une couleur différente de la couleur actuelle, et les messages de *Retour* ne sont pris en compte que s'ils ont une couleur semblable à le couleur courante.

5.3.3 Algorithme

L'algorithme $\mathcal{SG}$ utilise les deux variables binaires suivantes sur chaque processeur P_i :

1. *Couleur*$_i$: La couleur du nœud. Cette variable peut prendre les valeurs *noir* ou *blanc*, et peut être niée (en supposant que $\neg blanc = noir$ et que $\neg noir = blanc$).
2. *Phase*$_i$: La phase de la vague de communication courante. Cette variable peut prendre les valeurs *Diffusion* ou *Retour*.

Algorithme 2 *Le Synchroniseur Global pour le processeur P_i ($\mathcal{SG}$)*

Variables :

Couleur$_i$ *: la couleur*

Phase$_i$ *: la phase de la vague*

Constantes :

Enf$_i$ *: l'ensemble des enfants.*

Par$_i$ *: le parent.*

{Pour le processeur racine uniquement}

$\mathcal{SG} : \mathcal{R}_1$ *:* $(\forall k \in \text{Enf}_i,\ \text{Couleur}_k = \text{Couleur}_i \wedge \text{Phase}_k = \text{Retour}) \longrightarrow$

$\text{Couleur}_i \leftarrow \neg\text{Couleur}_i$ *;* $\text{Phase}_i \leftarrow \text{Diffusion}$

{Pour les processeurs intermédiaires uniquement}

$\mathcal{SG} : \mathcal{R}_2$ *:* $(\text{Couleur}_{\text{Par}_i} \neq \text{Couleur}_i) \longrightarrow$

$\text{Couleur}_i \leftarrow \text{Couleur}_{\text{Par}_i}$ *;* $\text{Phase}_i \leftarrow \text{Diffusion}$

{Pour les processeurs intermédiaires uniquement}

$\mathcal{SG} : \mathcal{R}_3$ *:* $(\forall k \in \text{Enf}_i,\ \text{Couleur}_k = \text{Couleur}_i \wedge \text{Phase}_k = \text{Retour}) \longrightarrow$

$\text{Phase}_i \leftarrow \text{Retour}$

{Pour les processeurs feuilles uniquement}

$\mathcal{SG} : \mathcal{R}_4$ *:* $(\text{Couleur}_{\text{Par}_i} \neq \text{Couleur}_i \vee \text{Phase}_i = \text{Diffusion}) \longrightarrow$

$\text{Couleur}_i \leftarrow \text{Couleur}_{\text{Par}_i}$ *;* $\text{Phase}_i \leftarrow \text{Retour}$

La figure 5.3 présente un exemple d'exécution de l'algorithme 2. Les flèches $\uparrow$ et $\downarrow$ placées à l'intérieur des nœuds dénotent la *Phase* du processeur (égale à *Retour* et *Diffusion*, respectivement). Les couleurs des nœuds *blanc* et *gris* dénotent la *Couleur* du processeur (égale à *blanc* et *noir*, respectivement). Un bord *gras* dénote que le processeur possède au moins une garde activable. Les flèches courbes pointillées dénotent l'exécution d'une règle gardée par un processeur.

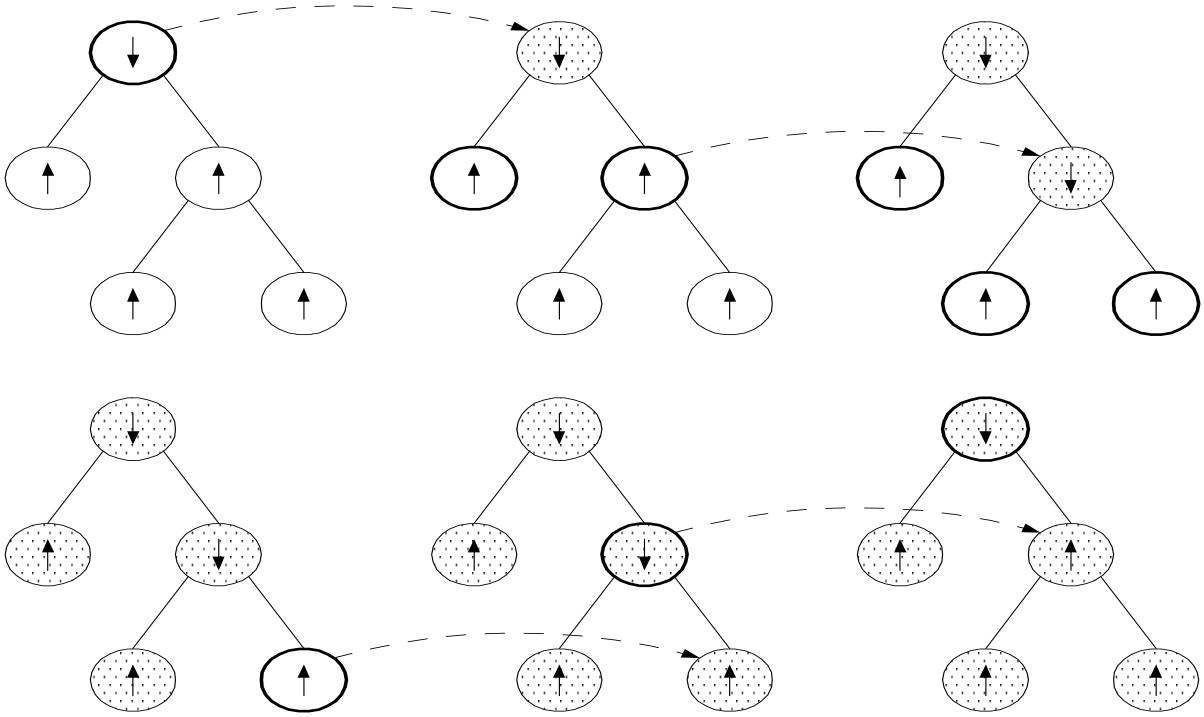

FIG. 5.3 – Exemple d'exécution de l'algorithme 2.

5.3.4 Preuve de Correction

Nous montrons tout d'abord qu'au bout d'un temps fini, tous les nœuds (excepté la racine) sont synchronisés avec leur parent. Rapellons que $\mathcal{C}$ et $\mathcal{E}$ dénotent respectivement l'ensemble des configurations et l'ensemble des exécutions d'un système réparti.

Lemme 5.1 (Synchronisation avec le Parent) $\forall c_1 \in \mathcal{C}$, $\forall e \in \mathcal{E}_{c_1}$, $e = c_1, c_2 \ldots$, $\forall j \in I \cup L$, $\exists i \geq 1$, $\text{Couleur}_j = \text{Couleur}_{\text{Par}_j}$ *dans* c_i.

Preuve: Supposons qu'il existe un nœud $j \in I$ tel que $Couleur_j \neq Couleur_{Par_j}$. Alors par la règle $\mathcal{SG} : \mathcal{R}_2$, $Couleur_j = Couleur_{Par_j}$. De manière similaire, supposons qu'il existe un nœud $j \in L$ tel que $Couleur_j \neq Couleur_{Par_j}$. Alors par la règle $\mathcal{SG} : \mathcal{R}_4$, $Couleur_j = Couleur_{Par_j}$. Par conséquent, aucun nœud intermédiaire ou feuille ne peut avoir indéfiniment une couleur différente de celle de son parent. □

Par la suite, nous démontrons la propriété de vivacité de l'algorithme $\mathcal{SG}$ en prouvant que la couleur du nœud racine ne peut rester indéfiniment la même

Lemme 5.2 (Vivacité) $\forall c_1 \in \mathcal{C}$, $\forall e \in \mathcal{E}_{c_1}$, Couleur_r *change infiniment souvent.*

Preuve: Supposons le contraire, c'est à dire qu'il existe une configuration $c_1 \in \mathcal{C}$, $\exists e \in \mathcal{E}_{c_1}$, c_r ne change jamais. Ceci implique que la règle $\mathcal{SG} : \mathcal{R}_0$ n'est jamais activable, et donc que $(\exists i \in Enf_r,\ Phase_i = Diffusion \vee Couleur_i \neq Couleur_r)$ est vraie indéfiniment. Par le lemme 5.1, le cas où $Couleur_i \neq Couleur_r$ indéfiniment est impossible. La condition $Phase_i = Diffusion$ signifie que $(\exists i' \in Enf_i,\ Phase_{i'} = Diffusion \vee Couleur_{i'} \neq Couleur_i)$ est vraie indéfiniment. Par induction sur la hauteur de l'arbre, cette proposition implique que $(\exists l \in L,\ Phase_l = Diffusion \vee Couleur_l \neq Couleur_{Par_l})$ est vraie indéfiniment. Alors, la règle $\mathcal{SG} : \mathcal{R}_4$ est activable, et lorsqu'elle est exécutée, nous avons $Phase_l = Retour$ et $Couleur_l = Couleur_{Par_l}$. Nous arrivons à une contradiction. □

Nous définissons maintenant un prédicat de synchronisation sur les configurations. Informellement, les configurations de $\mathcal{C}$ qui vérifient $\mathcal{L}^d_{\mathcal{SG}}$ sont celles dans lesquelles tous les nœuds jusqu'à distance d de la racine ont la même couleur et où tous les nœuds jusqu'à distance d de la racine (excepté la racine) ont leur phase égale à *concentration* (la racine ayant sa phase égale à *diffusion*).

Définition 5.2 (Prédicat de Synchronisation Globale) *Nous définissons le prédicat de synchronisation* $\mathcal{L}_{\mathcal{SG}}$ *sur* $\mathcal{C}$ *tel que* $\mathcal{L}_{\mathcal{SG}} = \mathcal{L}^h_{\mathcal{SG}}$ *où* h *est la hauteur de l'arbre et*

$$\mathcal{L}^d_{\mathcal{SG}} \equiv \left\{ \begin{array}{l} (\forall i \in Boule(r,d), \text{Couleur}_i = \text{Couleur}_r) \\ \wedge(\text{Phase}_r = \text{Diffusion}) \\ \wedge(\forall i' \in Boule(r,d) \setminus \{r\}, \text{Phase}_{i'} = \text{Retour}) \end{array} \right\}$$

Montrons maintenant que partant d'une configuration qui satisfait $\mathcal{L}_{\mathcal{SG}}$, toute exécution du système $\mathcal{SG}$ satisfait la spécification du synchroniseur global.

Lemme 5.3 (Correction) $\forall c_\alpha \in \mathcal{L}_{\mathcal{SG}}$, $\forall e \in \mathcal{E}_{c_\alpha}$, *e satisfait la spécification du synchroniseur global.*

Preuve: Tous les nœuds ont la même couleur $Couleur_{c_\alpha}$ dans c_α. Donc c_α est une configuration initiale valide pour $VC^{Couleur_{c_\alpha}}$. Afin d'établir le reste de la preuve, nous avons besoin de montrer qu'en un nombre fini de pas d'exécution, nous atteignons une autre configuration $c_\beta \triangleleft \mathcal{L}_{\mathcal{SG}}$ telle que $(\forall i \in Boule(r,h), Couleur_i = \neg\, Couleur_{c_\alpha}) \wedge (Phase_r = Diffusion) \wedge (\forall i' \in Boule(r,h) \setminus \{r\}, Phase_{i'} = Retour)$. L'unique règle activable dans c_α est $\mathcal{SG} : \mathcal{R}_1$ qui au moment de son exécution change $Couleur_r$ en $\neg\, Couleur_{c_\alpha}$. Dès lors, $\mathcal{SG} : \mathcal{R}_1$ n'est plus activable jusqu'à ce que tous les nœuds dans Enf_r aient la couleur $\neg Couleur_{c_\alpha}$ et soient dans la phase de retour. Les enfants de la racine ne peuvent changer leur phase en *Retour* avant qu'ils aient reçu un acquittement de chacun de leurs enfants. D'après les règles $\mathcal{SG} : \mathcal{R}_2$ et $\mathcal{SG} : \mathcal{R}_4$, les nœuds qui sont dans $I \cup L$ prennent la nouvelle couleur de leur parent $\neg Couleur_{c_\alpha}$, et la phase de retour peut être initiée seulement par les feuilles qui ont pris la nouvelle couleur. Quand tous les nœuds ont la même nouvelle couleur $\neg Couleur_{c_\alpha}$, seule la règle $\mathcal{SG} : \mathcal{R}_2$ peut être appliquée, continuant ainsi la phase de retour vers la racine. Après un temps fini, tous les nœuds possèdent la même couleur $\neg Couleur_{c_\alpha}$, et la racine est dans la phase de *Diffusion*, alors que tous les autres nœuds sont dans la phase de *Retour*. Cette dernière configuration satisfait le prédicat $\mathcal{L}_{\mathcal{SG}}$. □

Nous prouvons maintenant que pour toute configuration initiale et toute exécution possible de l'algorithme $\mathcal{SG}$, nous atteignons une configuration qui satisfait $\mathcal{L}_{\mathcal{SG}}$.

Lemme 5.4 (Convergence) $\forall c_1 \in \mathcal{C}$, $\forall e \in \mathcal{E}_{c_1}$, $e = c_1, c_2, \ldots$, $\exists i \geq 1$, $c_i \in \mathcal{L}_{\mathcal{SG}}$.

Preuve: Nous prouvons par induction la propriété suivante $P(k)$:

$$P(k) \equiv \forall c_1 \in \mathcal{C}, \forall e \in \mathcal{E}_{c_1}, e = c_1, c_2, \ldots, \exists i \geq 1, c_i \triangleleft \mathcal{L}^k_{\mathcal{SG}}$$

Cas de Base. Nous montrons que $P(1)$ est satisfaite. Considérons uniquement le nœud racine r et ses enfants. Nous avons besoin de considérer deux situations suivant qu'il existe un enfant avec une couleur différente de la racine ou non :

1. $\exists i \in Enf_r$, $Couleur_i \neq Couleur_r$. Comme il existe au moins un enfant avec une couleur différente de la sienne, la racine ne peut changer sa couleur (la règle $\mathcal{SG} : \mathcal{R}_1$ n'est pas activable). Mais, par le lemme 5.1, i change sa couleur en un temps fini.
2. $\forall i \in Enf_r$, $Couleur_i = Couleur_r$. Premièrement, comme tous les enfants ont la même couleur que leur parent, ils ne peuvent en changer (la règle $\mathcal{SG} : \mathcal{R}_2$ n'est pas activable et la règle $\mathcal{SG} : \mathcal{R}_4$ ne change pas la couleur). Deuxièmement, un enfant ne peut changer sa phase si celle-ci est déjà en mode *Retour* (le seul moyen de le faire est d'exécuter la règle $\mathcal{SG} : \mathcal{R}_3$, qui n'est pas activable).

   Maintenant considérons deux situations, dépendantes de la valeur de la phase des enfants :

(a) $\exists i \in Enf_r$, $Phase_i = Diffusion$. Ce type d'enfant ne peut être en phase de diffusion indéfiniment. Sinon, cela impliquerait que $\exists i' \in Enf_i$, $Phase_{i'} = Diffusion$ ou que $Couleur_{i'} \neq Couleur_i$ indéfiniment. Par induction sur la hauteur de l'arbre, ceci implique que $\exists l \in L$, $Phase_l = Diffusion$ ou que $Couleur_l \neq Couleur_{Par_l}$ indéfiniment. Alors la règle $\mathcal{SG} : \mathcal{R}_4$ est activable, et quand elle s'exécute la condition précédente devient fausse.

(b) $\forall i \in Enf_r$, $Phase_i = Retour$. Dans ce cas, $P(1)$ est satisfaite.

Etape d'Induction. Supposons qu'il existe un k $(1 \leq k \leq h-1)$ tel que $P(k)$ est satisfaite. Nous allons montrer que $P(k+1)$ est également satisfaite.

Puisque $P(k)$ est satisfaite, alors en partant de n'importe quelle configuration initiale, et pour toute exécution de $\mathcal{SG}$, il existe une configuration c_i telle que tous les nœuds jusqu'à la distance k de la racine ont la même couleur et également que ces nœuds (excepté la racine) ont leur phase égale à *Retour*. Dans la configuration c_i, l'unique règle applicable sur un nœud à distance inférieure ou égale à k de la racine est $\mathcal{SG} : \mathcal{R}_1$ (qui est activable à la racine). En exécutant $\mathcal{SG} : \mathcal{R}_1$, la racine change sa couleur. Par le lemme 5.1 et par induction sur k, tous les nœuds à distance au plus k prennent la nouvelle couleur.

Maintenant considérons qu'à distance *exactement* k de la racine, il existe des nœuds qui ne sont pas des feuilles (cet ensemble est forcément non-vide puisque la hauteur totale de l'arbre est *au moins* $k+1$). Nous notons cet ensemble de nœuds V^k. Ces nœuds possèdent déjà la même couleur que la racine et ont leur phase à *Diffusion* (ils ont exécuté la règle $\mathcal{SG} : \mathcal{R}_2$).

Par le lemme 5.1, les enfants des nœuds de V^k peuvent seulement prendre la couleur de leur parent et ne peuvent le faire avant que leur parent ait changé la sienne. Par des arguments similaires, ils ne peuvent changer leur couleur avant que la racine ait reçu des informations de *Retour* de la part de chacun de ses enfants. De même, les enfants des nœuds de V^k doivent changer leur phase à *Retour* et la garder. Fatalement, la phase de *Retour* continue par l'exécution de la règle $\mathcal{SG} : \mathcal{R}_3$. Comme tous les enfants des nœuds de V^k ont la même couleur et la même phase égale à *Retour*, $P(k+1)$ est satisfaite.

Nous avons montré que $P(1)$ est vraie, et que si $P(k)$ est vraie pour $1 \leq k \leq h-1$, alors $P(k+1)$ est également vraie. Par conséquent, $P(h)$ est vérifiée, et le lemme est prouvé. □

Finalement, nous montrons que le système $\mathcal{SG}$ est auto-stabilisant pour le problème de la synchronisation globale.

Théorème 3 *$\forall c_1 \in \mathcal{C}$, $\forall e \in \mathcal{E}_{c_1}$, $e = c_1, c_2, \ldots$, $\exists i \geq 1$, $\forall e' \in \mathcal{E}_{c_i}$, e' satisfait la spécification du synchroniseur global.*

Preuve: La preuve est une conséquence directe des lemmes 5.3 et 5.4. □

5.3.5 Complexité

Dans cette section, nous analysons à la fois la complexité en espace et en temps de l'algorithme $\mathcal{SG}$.

Proposition 5.1 (Complexité en Espace) *L'espace mémoire requis sur chaque processeur de l'algorithme $\mathcal{SG}$ est en $O(1)$.*

Preuve: L'algorithme 2 utilise seulement deux variables sur chaque processeur (sans compter la mémoire nécessaire à la construction de l'arbre)—*Couleur* et *Phase*. La variable *Phase* prend ses valeurs dans l'ensemble {*noir, blanc*}. La variable *Phase* prend ses valeurs dans l'ensemble {*Diffusion*, *Retour*}. Au total, 2 bits de mémoire sur chaque processeur sont suffisants. □

Proposition 5.2 (Complexité en Temps) *La complexité en temps T_s de l'algorithme $\mathcal{SG}$ est en $O(h)$, où h est la hauteur de l'arbre.*

Preuve: Soit $Couleur_r^i$ la couleur de la racine à l'instant T_i.

Soit T_0 le moment où toutes les défaillances cessent de se produire. Le moment situé à i unités de temps de T_0 est noté T_i. A l'instant T_1, tous les nœuds de $\Gamma^1(r)$ ont la couleur $Couleur_r^0$. Alors, à chacun des instants suivants, T_i $(2 \leq i \leq h)$, tous les nœuds de $\Gamma^i(r)$ ont la couleur $Couleur_r^0$. Entre les instants T_0 et T_h, deux situations peuvent se produire, suivant que la racine a changé sa couleur entre les instants T_0 et T_h :

1. La racine n'a pas changé sa couleur. Alors à l'instant T_h, tous les nœuds de $Boule(r, h)$ ont la couleur $Couleur_r^0$. Puisque h est la hauteur de l'arbre, tous les nœuds ont la même couleur à l'instant T_h. Donc, au temps T_{2h}, tous les nœuds de l'ensemble $I \cup L$ sont dans la phase de concentration tandis que la racine r est dans la phase de *Diffusion*, et tous les nœuds ont la couleur $Couleur_r^0$.
2. La racine a changé sa couleur à l'instant T_i $(1 \leq i \leq h)$ en $\neg\ Couleur_r^0$. Jusqu'à l'instant T_1, tous les nœuds de $\Gamma^1(r)$ avaient la couleur $Couleur_r^0$. De manière similaire, les nœuds de $\Gamma^2(r)$ avaient la couleur $Couleur_r^0$ jusqu'à l'instant T_2, et ainsi de suite. Comme il existe au moins une unité de temps entre les vagues $Couleur_r^0$ et $\neg Couleur_r^0$, la vague $\neg Couleur_r^0$ atteint seulement les nœuds colorés en $Couleur_r^0$. Alors, il est impossible qu'un nœud j, atteint par la vague $\neg Couleur_r^0$, soit dans la phase de *Retour* sauf si : (i) j est une feuille (ii) j est un nœud intermédiaire dont le sous-arbre est entièrement coloré avec $\neg Couleur_r^0$ et dont la phase vaut *Retour*. Donc, dès l'instant T_{i+h}, tous les nœuds ont la même couleur $\neg Couleur_r^0$, et dès l'instant $T_{i+2\times h}$, nous avons simultanément $(\forall j \in I \cup L,\ j \uparrow), (r \downarrow)$ et $(\forall i \in V,\ Couleur_i = \neg Couleur_r^0)$.

Dans tous les cas, en au plus $i + 2 \times h \leq 3 \times h$ unités de temps, l'arbre atteint une configuration qui satisfait le prédicat de sychronisation, $\mathcal{L}_{\mathcal{SG}}$ (Definition 5.2). □

Comparons la complexité en espace et en temps avec ceux de l'algorithme décrit dans [Var94] qui présente un schéma de synchronisation pour des réseaux en arbre. Le temps de stabilisation de [Var94] est en $O(h)$, ce qui est asymptotiquement identique à notre algorithme. Cependant, nous améliorons la complexité en espace — qui est en $O(\log n)$ dans [Var94] puisque celle de notre algorithme est en $O(1)$.

5.4 Le Synchroniseur de Voisinage

Dans cette section, nous présentons un moyen de synchroniser des processeurs voisins du graphe de communication $\mathcal{G}$ d'un système $\mathcal{S}$. Après avoir défini le problème à résoudre (section 5.4.1), nous donnons une solution informelle (section 5.4.2), puis un algorithme (l'algorithme $\mathcal{SV}$, section 5.4.3) que nous prouvons correct (section 5.4.4) et dont nous évaluons la complexité (section 5.4.5).

5.4.1 Spécification du Problème

Chacun des processeurs P_i possède une variable abstraite booléenne c_i. Dans toute exécution abstraite du système, les deux assertions suivantes sont vérifiées :

1. chaque processeur P_i change la valeur de sa variable c_i infiniment souvent (Vivacité) ;
2. entre deux changements successifs de valeurs de la variable c_i, tout voisin P_j de P_i a changé exactement une fois la variable c_j (Synchronisation de Voisinage).

5.4.2 Description Informelle

Chaque processeur possède une variable *couleur* pour indiquer le changement de son état à ses voisins. L'idée principale du synchroniseur de voisinage est la suivante : un processeur P_i change sa couleur uniquement quand il s'aperçoit que tous ses enfants (s'il en a) ont la même couleur que lui et que son parent (s'il en a) possède une couleur différente.

5.4.3 Algorithme

Le Synchroniseur de Voisinage ($\mathcal{SV}$) est décrit par l'algorithme 3. Chaque processeur P_i possède une variable $Couleur_i$, qui détermine son état.

Algorithme 3 *Synchroniseur de Voisinage pour le processeur P_i (SV)*

Variable :

Couleur_i *: La variable couleur.*

Constantes :

Enf_i *: L'ensemble des enfants.*

Par_i *: Le parent.*

Actions :

{Pour le processeur racine}

$\mathcal{SV} : \mathcal{R}_1 :: \forall j \in \text{Enf}_i, \text{Couleur}_j = \text{Couleur}_i \longrightarrow$

$\text{Couleur}_i \leftarrow \neg\text{Couleur}_i$

{Pour les processeurs intermédiaires}

$\mathcal{SV} : \mathcal{R}_2 :: \text{Couleur}_{\text{Par}_i} \neq \text{Couleur}_i \wedge (\forall j \in \text{Enf}_i, \text{Couleur}_j = \text{Couleur}c_i) \longrightarrow$

$\text{Couleur}_i \leftarrow \text{Couleur}_{\text{Par}_i}$

{Pour les processeurs feuilles}

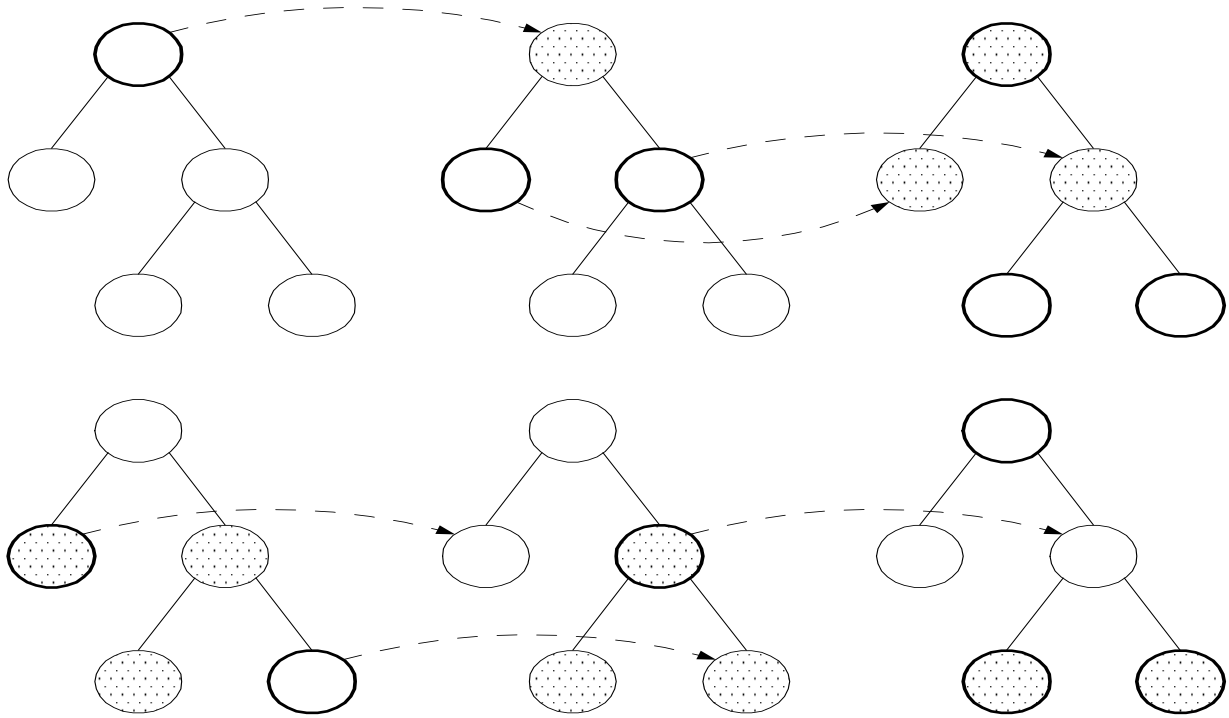

FIG. 5.4 – Un exemple d'exécution de l'algorithme 3.

$\mathcal{SV} : \mathcal{R}_3 :: \text{Couleur}_{\text{Par}_i} \neq \text{Couleur}_i \longrightarrow$
$\text{Couleur}_i \leftarrow \text{Couleur}_{\text{Par}_i}$

Exemple d'Exécution La figure 5.4 présente un exemple d'exécution de l'algorithme 3. Les couleurs blanc et gris dénotent une couleur de processeur blanche et noire respectivement. Un bord gras dénote que le processeur possède au moins une garde activable. Les flèches courbes pointillées dénotent l'exécution d'une règle gardée.

5.4.4 Preuve de Correction

Nous prouvons tout d'abord la vivacité de l'algorithme, puis la correction de chacune des exécutions de l'algorithme. La propriété suivante découle directement de la lecture de l'algorithme 3.

Propriété 5.1 $\forall c \in \mathcal{C}, \forall i \in V, \text{Activable}(i, c) \Rightarrow (\forall j \in \Gamma^1_{\mathcal{G}}(i), \neg\text{Activable}(j, c))$.

Lemme 5.5 *Pour tout nœud* $i \in (\{r\} \cup I)$*, l'assertion suivante est vérifiée :* $\forall c \in \mathcal{C}, \exists c\prime \in \mathcal{C}, c \rightsquigarrow c\prime, \forall j \in \text{Enf}_i, \text{Couleur}_j = \text{Couleur}_i$ *dans la configuration* $c\prime$.

Preuve: Supposons que P_i exécute une action ($\mathcal{SV} : \mathcal{R}_1$ ou $\mathcal{SV} : \mathcal{R}_2$) durant une exécution e. Alors, tous les enfants de P_i doivent avoir la même couleur que lui avant qu'il puisse exécuter son action (voir la garde de $\mathcal{SV} : \mathcal{R}_1$ et $\mathcal{SV} : \mathcal{R}_2$).

Considérons maintenant le cas où P_i n'exécute jamais d'action pendant l'exécution e (c'est à dire que P_i ne change jamais sa couleur). Une fois qu'un enfant P_j de P_i possède une couleur identique à celle de P_i, P_j n'est plus en mesure d'exécuter une action. Nous prouvons ceci par récurrence sur la hauteur du sous-arbre enraciné en i.

(i) **Cas de base :** $h_i = 1$, c'est à dire que i est le parent d'une feuille. Supposons qu'il existe un nœud $j \in Enf_i$ tel que $Couleur_j$ ne devient jamais égal à $Couleur_i$ pendant une exécution e. Alors, dans toutes les configurations de e, $\mathcal{SV} : \mathcal{R}_3$ demeure activable jusqu'à ce que P_j exécute $\mathcal{SV} : \mathcal{R}_3$. Par équité, P_j va fatalement exécuter $\mathcal{SV} : \mathcal{R}_3$ et $Couleur_j = Couleur_i$ devenir vrai. Finalement, tous les processeurs enfants de P_i vont prendre sa couleur.

(ii) **Etape d'induction :** Supposons que le lemme soit vrai pour $0 < h_i \leq m, m \leq h-1$. Supposons qu'il existe deux processeurs P_i et P_j tels que $h_i = m+1, j \in Enf_i$ et P_j ne prend jamais la couleur de P_i au cours de l'exécution e. Alors P_j ne peut exécuter $\mathcal{SV} : \mathcal{R}_2$ car cela contredirait l'hypothèse $Couleur_j = Couleur_i$. Par l'hypothèse de récurrence, le système atteindra une configuration γ où les enfants de P_j obtiendront la couleur de P_j. Dès lors, $Activable(\mathcal{SV} : \mathcal{R}_2, j, c)$ demeurera vrai jusqu'à ce que P_j exécute $\mathcal{SV} : \mathcal{R}_2$ (par la propriété 5.1). Par équité, P_j va fatalement exécuter $\mathcal{SV} : \mathcal{R}_2$ et $Couleur_j = Couleur_i$ devenir vrai. Une fois que $Couleur_j = Couleur_i$, P_j ne peut plus changer sa couleur. Par conséquent, tous les enfants de P_i vont prendre la couleur de P_i.

□

Nous sommes maintenant en mesure de prouver la vivacité de l'algorithme $\mathcal{SV}$.

Lemme 5.6 (Vivacité) $\forall e \in \mathcal{E}, \forall i \in V$, *$P_i$ exécute une action infiniment souvent.*

Preuve: Prouvons ceci par contradiction. Supposons qu'il existe au moins un processeur qui cesse d'exécuter une action à partir de la configuration c lors de l'exécution e. Soit i l'un des processeurs les plus proches de la racine parmi ces processeurs. En vertu du lemme 5.5, dans une configuration $c\prime$, $c \rightsquigarrow c\prime$, tous les processeurs enfants de P_i possèdent la même couleur que P_i. Comme P_i ne peut changer sa propre couleur, aucun des processeurs enfants de P_i ne peut changer de couleur à partir de la configuration $c\prime$ dans l'exécution e.

(i) Si $i = r$, alors $Activable(\mathcal{SV} : \mathcal{R}_1, i, c\prime)$ est vrai. Par équité, P_i va fatalement exécuter $\mathcal{SV} : \mathcal{R}_1$.

(ii) Si $i \neq r$ et $Couleur_i \neq Couleur_{Par_i}$, alors soit $Activable(\mathcal{SV} : \mathcal{R}_2, i, c\prime)$ (dans le cas où $i \in I$) soit $Activable(\mathcal{SV} : \mathcal{R}_3, i, c\prime)$ (dans le cas où $i \in L$) est vrai. Par équité, P_i va fatalement exécuter $\mathcal{SV} : \mathcal{R}_2$ ou $\mathcal{SV} : \mathcal{R}_3$.

(iii) Si $i \neq r$ et $Couleur_i = Couleur_{Par_i}$. Par_i va fatalement exécuter une action afin de changer sa couleur car par hypothèse, tous les ancêtres de P_i changent leur couleur infiniment souvent. Après que Par_i a exécuté cette action, $Couleur_i \neq Couleur_{Par_i}$ est

vrai. Dès lors, Par_i ne peut changer sa couleur à nouveau puisque P_i ne la change plus. Donc, par équité, P_i va fatalement exécuter $\mathcal{SV} : \mathcal{R}_2$ ou $\mathcal{SV} : \mathcal{R}_3$.

□

Lemme 5.7 (Synchronisation) *Soit $\mathcal{SV} : \mathcal{R}_i$ une action exécutée par le processeur P_i. $\forall e \in \mathcal{E}, \forall i \in (\{r\} \cup I), \forall j \in \text{Enf}_i$, la projection sur e des actions de P_i et P_j peut être représentée par l'expression suivante : $(\mathcal{SV} : \mathcal{R}_i; \mathcal{SV} : \mathcal{R}_j)^\omega \cup (\mathcal{SV} : \mathcal{R}_j; \mathcal{SV} : \mathcal{R}_i)^\omega$.*

Cette dernière expression se lit comme :
(i) Entre deux actions quelconque de P_i (respectivement P_j), P_j (respectivement P_i) exécute exactement une action ;
(ii) P_i et P_j executent des actions infiniment souvent.

Preuve: Directe par les lemmes 5.6 et 5.5. □

Théorème 4 (Auto-stabilisation) *L'algorithme 3 est auto-stabilisant.*

Preuve: Le lemme 5.7 prouve qu'entre deux actions quelconque exécutées par un processeur, chacun de ses voisins exécute exactement une action. Cela signifie que toute exécution de l'algorithme 3 est correcte. Comme toute configuration initiale suffit aux hypothèses du lemme 5.7, les propriétés de clôture et de convergence sont trivialement satisfaites, signifiant que l'algorithme 3 est auto-stabilisant. □

5.4.5 Complexité

L'algorithme $\mathcal{SV}$ utilise une unique variable booléenne *Couleur*. Les constantes *Enf* et *Par* ne sont pas prises en compte puisqu'elles sont soit stockées dans le code du processeur si le réseau est effectivement un arbre, soit maintenues par un algorithme sous-jacent de construction d'arbre. Par conséquent, l'algorithme $\mathcal{SV}$ requiert seulement un espace mémoire de 1 bit sur chacun des processeurs du réseau.

Puisque chacune des exécutions partant d'une configuration quelconque du système satisfait la spécification, l'algorithme $\mathcal{SV}$ possède un temps de stabilisation qui est nul, donc optimal.

5.5 Applications

Dans cette section nous proposons plusieurs applications pour nos algorithmes de synchronisation auto-stabilisants. Trois applications directes du synchroniseur global sont données dans les sections 5.5.1 (horloge globale), 5.5.2 (calcul global) et 5.5.3 (tri réparti) et ne sont qu'informellement décrites. Une application montrant le gain en parallélisation du synchroniseur de voisinage est présentée en détail dans la section 5.5.4 (diffusion parallèle).

5.5.1 Horloge Globale

Nous montrons comment utiliser l'algorithme 2 pour construire un algorithme émulant une horloge globale numérique. Bien que bâti à partir d'un canevas général, cet algorithme améliore la meilleure complexité en mémoire connue tout en maintenant une temps de stabilisation asymptotiquement optimal.

Spécification Informelle du Problème

Les processeurs maintiennent une variable *horloge*, qui peut représenter le compteur de programme d'un algorithme externe exécutant indéfiniment un séquence de k actions :

$$a_0, a_1, \ldots, a_{k-1}, a_0, a_1, \ldots$$

Il est alors demandé à tous les nœuds d'effectuer la même action de l'algorithme externe "simultanément". En fait, les horloges de tous les nœuds doivent être les même quand les processeurs effectuent les actions de l'algorithme externe, et doivent s'incrémenter à l'unisson.

Description Informelle

Nous ajoutons une variable *horloge* à chaque processeur du réseau arborescent. L'algorithme 2 est légèrement modifié de la manière suivante :

1. Les processeurs intermédiaires et feuilles prennent la valeur de l'horloge de leur parent en même temps qu'ils changent de couleur.
2. Le processeur racine incrémente sa valeur d'horloge modulo k (où k désigne le nombre de pas de calcul de l'algorithme externe) au moment où il change sa couleur.

En un temps $O(h)$, le système atteint une configuration synchronisée par la couleur – les variables d'horloge de tous les nœuds ont alors la même valeur. En partant de cette configuration, à chaque fois que le processeur racine initie une nouvelle vague colorée (en utilisant une couleur différente), il incrémente sa variable d'horloge. Par conséquent, après un temps $O(h)$, les variables d'horloge sont synchronisées. Ce temps de stabilisation a été montré asymptotiquement optimal dans [AKM+93].

La complexité en espace mémoire de cet algorithme de synchronisation globale modifié est toujours en $O(1)$ (ou en $O(\delta)$ si la mémoire nécessaire à maintenir la topologie arborescente est prise en compte), puisque la variable d'horloge utilise un espace mémoire en $O(\log k)$ et que k est fixé quelle que soit la taille du réseau. Ceci constitue une amélioration par rapport à d'autres algorithmes *asynchrones* d'Unisson ou de synchronisation d'horloge tels que ceux décrits dans [AKM+93] (qui nécessite un espace mémoire en $O(\log n \times \log D)$), dans [ADG91] (qui utilise un espace mémoire en $O(\log n)$), ou dans [CFG92] (qui utilise un espace mémoire en $O(\log n^2)$. Les algorithmes *synchrones* de synchronisation d'horloge comme ceux décrits dans [CS94], [Dol97] et [HG95] nécessitent au moins un espace mémoire en $O(1)$ et ont un temps de stabilisation en $O(h)$.

5.5.2 Fonction Globale

Nous mentionnons ici des résultats décrits en détail dans [RH90] et [Tel94] concernant le calcul de fonctions globales au moyen d'un synchroniseur global.

Spécification Informelle du Problème

Il est demandé à chacun des nœuds du système de collaborer afin d'effectuer un calcul global sur des variables qui sont réparties sur un réseau. Nous demandons à l'opération devant être effectuée d'être *associative* et *commutative*. De telles opérations incluent le calcul du *Maximum*, du *Minimum*, de la *Somme* des variables de tous les processeurs, etc. Après stabilisation du système, chacun des processeurs doit détenir le résultat correct du calcul.

Description Informelle

Soit f une opération commutative et associative devant être effectuée sur un réseau. Nous utilisons le schéma suivant pour calculer f :

- Les phases de *Retour* sont utilisées pour calculer l'opération f.
- Les phases de *Diffusion* sont utilisées pour diffuser le résultat de l'opération f précédente effectuée sur tous les nœuds du réseau.

Nous ajoutons trois variables à chacun des processeurs : *valeur*, qui contient la valeur sur laquelle l'opération f doit être effectuée, *résultatLocal* qui contient le résultat (appelé résultat *local*) de l'opération f sur le sous-arbre enraciné sur le nœud, et *résultatGlobal* qui contient le résultat (appelé résultat *global*) de l'opération f précédemment effectuée sur la totalité du réseau.

L'algorithme 2 est légèrement modifié de la manière suivante :

- **Calcul de f** : Les processeurs feuilles calculent f en utilisant leur propre *valeur*, et inscrivent le résultat dans leur variable *résultatLocal* et mettent leur type de phase à la valeur *Retour*. Les processeurs intermédiaires et le processeur racine calculent f à partir des variables *résultatLocal* des processeurs contenus dans le sous-arbre dont ils sont la racine et de leur propre *valeur*, inscrivent le résultat dans leur variable *résultatLocal* et mettent leur type de phase à la valeur *Retour*.
- **Diffusion du Résultat** : Le processeur racine copie sa variable *résultatLocal* dans sa variable *résultatGlobal* au moment où il change sa couleur. Les processeurs feuilles et intermédiaires copient la valeur de la variable *résultatGlobal* de leur processeur parent au moment où ils copient leur couleur.

5.5.3 Tri Réparti

Dans cette section, nous utilisons une combinaison des applications d'horloge globale et de fonction globale pour effectuer de manière auto-stabilisante un tri sur des valeurs réparties sur les processeurs suivant le rang de leur identifiant dans le système : le processeur dont

l'identifiant est le plus petit obtient la plus petite valeur, le processeur dont l'identifiant est immédiatement supérieur obtient une valeur supérieure ou égale, et ainsi de suite.

Spécification du Problème

Chaque processeur du système dispose d'un identifiant unique sur le réseau sur lequel peut être définie une relation d'ordre total. Par ailleurs, il possède trois variables :

1. *valeurEntrée* : variable entière dont le contenu est mis à jour par l'utilisateur du système ;
2. *valeurSortie* : variable entière.

Le but de l'algorithme de tri réparti est de faire en sorte qu'après stabilisation du système, les variables *valeurSortie* de chacun des processeurs contiennent une permutation des valeurs des variables *valeurEntrée* telle que les rangs respectifs des *valeurSortie* et des identifiants soient les mêmes.

Description Informelle

Nous construisons un système où une horloge globale permet de distinguer $N+1$ phases de diffusion différentes, où N est le nombre de processeurs du système. Chaque phase de *Retour* numéro i est chargée de calculer le $i^{ème}$ plus petit identifiant du système, ainsi que la $i^{ème}$ plus petite *valeurEntrée* du système. Chaque phase de *Diffusion* numéro $i+1$ est chargée d'indiquer à chaque processeur quels est le $i^{ème}$ plus petits identifiant et sa *valeurEntrée* associée.

Chaque processeur, lors d'une phase de *Diffusion*, place la *valeurEntrée* associée dans sa propre *valeurSortie* si et seulement si l'identifiant de processeur diffusé est le sien. Lors d'une phase de *Retour*, chaque processeur retourne le plus petit identifiant et la plus petite *valeurEntrée* située dans son sous-arbre et s'inclut lui-même s'il n'a pas déjà été choisi lors d'une phase de *Diffusion* précédente.

Le fait d'avoir déjà été choisi peut être mémorisé au moyen d'une variable booléenne, mais cette variable peut être modifiée incorrectement suite à une défaillance transitoire. Aussi la phase de *Diffusion* numéro 0 réinitialise toutes ces variable booléennes à *faux*. Les développements techniques de cette application peuvent être trouvés dans [ABC+98].

5.5.4 Diffusion Parallèle

Dans cette section, nous utilisons l'algorithme $\mathcal{SV}$ comme brique de base à la construction d'un algorithme auto-stabilisant de diffusion sur un réseau arborescent. Informellement, le problème à résoudre s'enonce comme suit : la racine dispose d'une séquence infinie de messages à diffuser à tous les processeurs de l'arbre. La racine diffuse ses messages en parallèle, c'est à dire qu'elle n'attend pas d'avoir reçu un acquittement de chacun des processeurs du réseau, mais qu'elle attend seulement que ses propres enfants aient acquitté la réception du message. Par conséquent, à tout instant, plusieurs messages différents peuvent se propager

de la racine aux feuilles, implantant de fait un mécanisme de pipeline. Nous montrons dans la section 5.5.4 quels sont les gains en efficacité de cet algorithme consécutifs de la propagation concurrente des messages.

Dans la suite, nous définissons tout d'abord le problème, puis nous montrons de manière informelle comment transformer l'algorithme $\mathcal{SV}$ afin d'obtenir un algorithme de diffusion parallèle ($\mathcal{DP}$). Puis nous montrons que l'algorithme $\mathcal{DP}$ est auto-stabilisant.

Spécification du Problème

Nous notons $Dist(r, i)$ la distance (en nombre d'arètes) d'un nœud i à la racine r. Nous considérons qu'une exécution e satisfait la spécification $\mathcal{SP}_{\mathcal{DP}}$ de la tâche de la Diffusion si les conditions suivantes sont vérifiées :

(i) Tout message émis par la racine est fatalement reçu par tous les processeurs de l'arbre dans l'ordre où ils ont été émis. Cette propriété est appelée *Acheminement Correct*.

(ii) Tous les messages, sauf (éventuellement) les premiers $Dist(r, i)$ messages reçus par le processeur P_i, ont été émis par le processeur racine. Cette propriété est appelée *Validité de Contenu*.

(iii) L'algorithme $\mathcal{DP}$ est auto-stabilisant.

Description Informelle

La synchronisation de voisinage peut être appliquée dès qu'il est besoin de simuler un mécanisme de passage de message fiable dans un modèle à registres. La solution de base est comme suit : le processeur racine envoie un nouveau message à ses enfants et attend que tous ses enfants aient pris connaissance du message. Quand cela se produit, le processeur racine peut envoyer un nouveau message. Un processeur interne prend connaissance d'un message de son parent uniquement quand il s'aperçoit que tous ses enfants ont pris connaissance du message précédent. Les processeurs feuilles lisent simplement tous les nouveaux messages en provenance de leur parent. Le processeur racine utilise le changement de sa variable de couleur pour indiquer qu'un nouveau message est à diffuser. Les processeurs internes et feuilles changent leur couleur pour indiquer à leur parents qu'ils ont pris connaissance du message précédent, et pour indiquer à leurs enfants qu'un nouveau message est à diffuser.

Algorithme

Quelques modifications simples suffisent à transformer l'algorithme $\mathcal{SV}$ en un nouvel algorithme $\mathcal{DP}$. Chaque processeur P_i se voit doté d'une variable supplémentaire m_i servant à stocker le message courant reçu du processeur parent. Le processeur racine P_r lit un nouveau message utilisateur et l'écrit dans sa variable m_r. Les processeurs internes et les processeurs feuilles dupliquent le message de leur parent contenu dans m_{Par_i} vers leur propre variable m_i. L'algorithme de diffusion $\mathcal{DP}$ est le suivant :

Algorithme 4 *Diffusion pour le processeur P_i ($\mathcal{DP}$)*

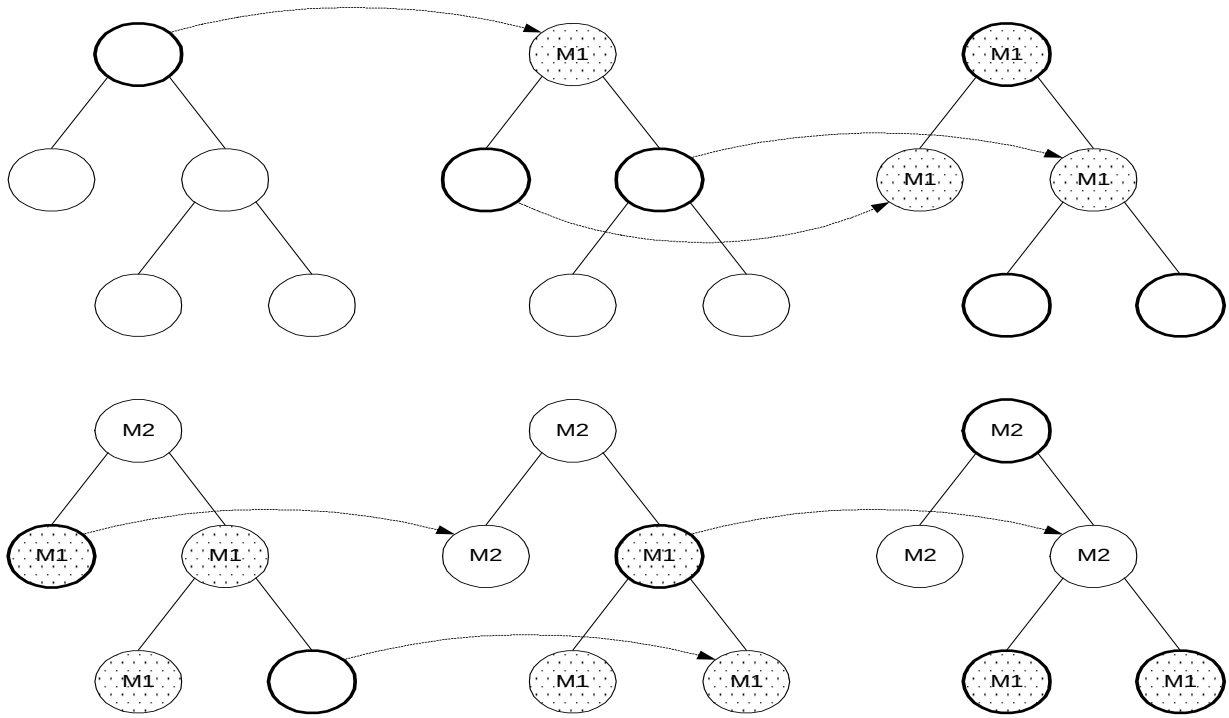

FIG. 5.5 – Un exemple d'exécution de l'algorithme 4.

Variables :

Couleur_i *: La variable* couleur*.*

m_i *: La variable* message*.*

Constantes :

Enf_i *: L'ensemble des enfants.*

Par_i *: Le parent.*

Actions :

{Pour le processeur racine}

$\mathcal{DP} : \mathcal{R}_1 : (\forall j \in \text{Enf}_i, \text{Couleur}_j = \text{Couleur}_i) \longrightarrow$

$m_i \leftarrow <$*message suivant*$>$; $\text{Couleur}_i \leftarrow \neg\text{Couleur}_i$

{Pour les processeurs internes}

$\mathcal{DP} : \mathcal{R}_2 : (\text{Couleur}_{\text{Par}_i} \neq \text{Couleur}_i) \wedge (\forall j \in \text{Enf}_i, \text{Couleur}_j = \text{Couleur}_i) \longrightarrow$

$m_i \leftarrow m_{\text{Par}_i}$; $\text{Couleur}_i \leftarrow \text{Couleur}_{\text{Par}_i}$

{Pour les processeurs feuilles}

$\mathcal{DP} : \mathcal{R}_3 : (\text{Couleur}_{\text{Par}_i} \neq \text{Couleur}_i) \longrightarrow$

$m_i \leftarrow m_{\text{Par}_i}$; $\text{Couleur}_i \leftarrow \text{Couleur}_{\text{Par}_i}$

La figure 5.5 présente un exemple d'exécution de l'algorithme 4. Les messages successifs émis par la racine sont indiqués par M_1, M_2 et M_3. Les couleurs blanc et gris dénotent une

couleur de processeur blanche et noire respectivement. Un bord gras dénote que le processeur possède au moins une garde activable. Les flèches courbes pointillées dénotent l'exécution d'une règle gardée.

Preuve de Correction

Lemme 5.8 $\forall i \in (\{r\} \cup I), \forall j \in \text{Enf}_i$, *les messages envoyés par* P_i *sont reçus par* P_j *dans l'ordre où ils ont été émis, sans perte ni duplication.*

Preuve: Par le lemme 5.7 et le code de l'algorithme 4, après que P_i a reçu un message, il ne peut exécuter son action avant que tous ses processeurs enfants n'aient exécuté leur action (et lu le message m_i en provenance de P_i). □

Un corollaire immédiat s'ensuit :

Corollaire 5.1 $\forall i \in (I \cup L)$, *tous les messages, éventuellement excepté le premier, reçus par* P_i *ont été émis par* Par_i.

Notons que les premiers messages reçus par le processeur P_i peuvent n'avoir pas été émis ou reçus par Par_i car ces messages pouvaient être corrompus suite à des défaillances transitoires.

Lemme 5.9 (Acheminement Correct) *Tous les messages émis par la racine sont fatalement reçus par tous les processeurs de l'arbre dans l'ordre où ils ont été émis, sans perte ni duplication.*

Preuve: La preuve découle du lemme 5.8 et d'une induction sur la hauteur de l'arbre. □

Lemme 5.10 (Validité de Contenu) *Tous les messages, sauf (éventuellement) les premiers* $Dist(r, i)$ *messages reçus par le processeur* P_i *ont été émis par le processeur racine.*

Preuve: La preuve découle du corollaire 5.1 et d'une induction sur $Dist(r, i)$, la distance du nœud i à la racine. □

Théorème 5 *L'algorithme 4 est auto-stabilisant.*

Preuve: D'après les lemmes 5.9 et 5.10, en partant d'une configuration initiale quelconque, tout exécution satisfait à la fois les prédicats d'acheminement correct et de validité de contenu. Donc l'algorithme $\mathcal{DP}$ est correct. Comme toutes les exécutions satisfont le prédicat $\mathcal{SP}_{\mathcal{DP}}$, l'algorithme $\mathcal{DP}$ satisfait trivialement les propriétés de clôture et de convergence, et est par conséquent auto-stabilisant. □

Complexité

Dans cette section, nous calculons les prérequis en espace mémoire et en temps de stabilisation de l'algorithme $\mathcal{DP}$, ainsi que le temps nécessaire à la diffusion de m messages depuis le processeur racine dans tout le réseau arborescent.

Complexité en Espace. L'algorithme $\mathcal{DP}$ utilise deux variables : *Couleur* et m. Puisque m est utilisé uniquement pour transmettre des messages de la surcouche applicative, l'espace additionnel requis par notre algorithme est de seulement 1 bit.

Complexité en Temps de Stabilisation. Comme l'indique la preuve de correction de l'algorithme $\mathcal{DP}$, toute exécution partant d'une configuration quelconque est correcte vis à vis de la spécification $\mathcal{SP}_{\mathcal{DP}}$. Par suite, il est trivial d'en déduire un temps de stabilisation nul, et donc en $O(1)$.

Temps de Diffusion. Afin de calculer le temps nécessaire à la diffusion de m messages, nous devons d'abord prouver quelques propriétés. Nous définissons tout d'abord ce que cela signifie pour un processeur d'être *prêt*.

Définition 5.3 *Un processeur P_i est* prêt *si au moins une des conditions suivantes est vraie :* (i) $i \in (\{r\} \cup L)$, (ii) $\text{Couleur}_i = \text{Couleur}_{\text{Par}_i}$ *ou* (iii) $\forall j \in \text{Enf}_i$, $\text{Couleur}_i = \text{Couleur}_j$.

L'ensemble des configurations où tous les processeurs sont prêts est dénotée par $\mathcal{L}_{cs}$.

Lemme 5.11 *Soit P_i un processeur tel que P_i et ses processeurs enfants soient prêts. Après un cycle asynchrone d'exécution, P_i est toujours prêt.*

Preuve: Considérons un nœud $i \in I$. Nous n'avons par besoin de considérer les processeurs racine et feuilles, puisque par définition, ils sont toujours prêts.

(i) Supposons que P_i change sa *Couleur* pendant ce cycle asynchrone d'exécution en copiant la *Couleur* de son processeur parent. Par la propriété 5.1, le processeur parent et les processeurs enfants de P_i ne peuvent exécuter d'action pendant ce cycle asynchrone d'exécution. En tout état de cause, P_i demeure prêt puisque la condition *(ii)* de la définition 5.3 est satisfaite.

(ii) Supposons que P_i ne change pas sa *Couleur* pendant ce cycle asynchrone d'exécution. Soit j un nœud enfant de i et tel que $Couleur_j \neq Couleur_i$ avant le début du cycle asynchrone d'exécution (dans une configuration c). Puisque P_j est prêt dans la configuration c, P_j doit satisfaire la condition *(iii)* de la définition 5.3, c'est à dire que tous les processeurs enfants de P_j ont la même *Couleur* que P_j dans la configuration c. Donc, l'une au moins des règles $\mathcal{DP} : \mathcal{R}_2$ et $\mathcal{DP} : \mathcal{R}_3$ est activable par P_j. Dans ce cycle asynchrone d'exécution, P_j exécute son action et $Couleur_j$ devient égal à $Couleur_i$. Soit k un nœud enfant de i et tel que $Couleur_k = Couleur_i$ dans la configuration c. Le processeur P_k ne peut exécuter son action durant ce cycle asynchrone d'exécution. La condition *(iii)* de la définition 5.3 est satisfaite sur le processeur P_i après le cycle asynchrone d'exécution

(iii) Supposons que Par_i change sa *Couleur* durant ce cycle asynchrone d'exécution. Par la propriété 5.1, P_i ne peut exécuter son action durant ce cycle asynchrone d'exécution (voir le cas *(i)*).

□

Corollaire 5.2 . *$\forall c \in \mathcal{L}_{cs}$, $\forall e \in \mathcal{E}_c$, la configuration c' atteinte après un cycle asynchrone d'exécution à partir de la configuration c est telle que $c' \in \mathcal{L}_{cs}$.*

Lemme 5.12 *En partant d'une configuration quelconque, après au plus $h-1$ cycles asynchrones d'exécution, le système atteint une configuration $c \in \mathcal{L}_{cs}$.*

Preuve: Nous prouvons ce lemme par induction sur h_i, la hauteur du sous-arbre enraciné sur le nœud i.

(i) **Cas de Base :** $h_i = 0$. Le lemme est vrai pour $h_i = 0$ (les processeurs feuilles) par définition.

(ii) **Etape d'Induction :** Supposons que lemme soit vrai pour $0 \leq h_i \leq m, m \leq h-2$. Notons que ce lemme est vrai pour $h_i = h$ puisque la racine est toujours prête. Les processeurs qui sont les racine d'un sous-arbre de hauteur inférieure ou égale à m sont prêts après m cycles asynchrones d'exécution (hypothèse d'induction) et le demeurent par la suite (corollaire 5.2). Nous avons maintenant besoin de prouver que ce lemme est également vrai pour $h_i = m+1$. Soit P_i un processeur tel que $h_i = m+1$. Supposons que P_i ne soit pas prêt après m cycles asynchrones d'exécution et qu'il existe un nœud $j \in Enf_i$ tel que $Couleur_j \neq Couleur_i$ avant le $m+1^{\text{ième}}$ cycle asynchrone d'exécution. Le processeur P_i peut changer de couleur durant le $m+1^{\text{ième}}$ cycle asynchrone d'exécution. Si le nœud k est un nœud enfant de i et tel que $Couleur_k = Couleur_i$ avant le $m+1^{\text{ième}}$ cycle asynchrone d'exécution, alors le processeur P_k ne peut exécuter d'action durant $m+1$ cycles asynchrones d'exécution. Soit le nœud j un nœud enfant de i et tel que $Couleur_j \neq Couleur_i$ avant le $m+1^{\text{ième}}$ cycle asynchrone d'exécution. Comme $h_j \leq m$, le processeur P_j est prêt en au plus m cycles asynchrones d'exécution par hypothèse d'induction. Tous les processeurs enfants de P_j prennent la couleur de ce dernier après m cycles asynchrones d'exécution. Donc, l'une au moins des règles $\mathcal{DP} : \mathcal{R}_2$ et $\mathcal{DP} : \mathcal{R}_3$ est activable par le processeur P_j. Au cours du $m+1^{\text{ième}}$ cycle asynchrone d'exécution, le processeur P_j exécute son action et $Couleur_j$ prend la valeur de $Couleur_i$.

(iii) **Conclusion :** Après h_i cycles asynchrones d'exécution, tous les processeurs enfants de P_i partagent sa couleur, et i est prêt.

□

Définition 5.4

$$\mathcal{L}^d_{even} \equiv \forall k \in]0,d], \forall i \in V,\ Dist(r,i) = 2k,\ \ \text{Couleur}_i = \text{Couleur}_{P_i}$$

Une configuration c satisfait $\mathcal{L}^d_{even}$ si tous les processeurs à distance paire et inférieure ou égale à $2 \times d$ de la racine ont la même Couleur *que leur processeur parent.*

$$\mathcal{L}^d_{odd} \equiv \forall k \in [0,d], \forall i \in V,\ Dist(r,i) = 2k+1,\ \ \text{Couleur}_i = \text{Couleur}_{P_i}$$

Une configuration c satisfait $\mathcal{L}^d_{odd}$ *si tous les processeurs à distance impaire et inférieure ou égale à* $2 \times d + 1$ *de la racine ont la même* Couleur *que leur processeur parent.*

$$\mathcal{G}^d_{even} \equiv \forall k \in]0, d], \forall i \in V,\ Dist(r, i) = 2k,\ \text{Couleur}_i \neq \text{Couleur}_{P_i}$$

Une configuration c satisfait $\mathcal{G}^d_{even}$ *si tous les processeurs à distance paire et inférieure ou égale à* $2 \times d$ *de la racine n'ont pas la même* Couleur *que leur processeur parent.*

$$\mathcal{G}^d_{odd} \equiv \forall k \in [0, d], \forall i \in V,\ Dist(r, i) = 2k + 1,\ \text{Couleur}_i \neq \text{Couleur}_{P_i}$$

Une configuration c satisfait $\mathcal{G}^d_{odd}$ *si tous les processeurs à distance impaire et inférieure ou égale à* $2 \times d + 1$ *de la racine n'ont pas la même* Couleur *que leur processeur parent.*

$\mathcal{G}^0_{even} \equiv \mathcal{L}_{cs}$. $\mathcal{L}_{even} \equiv \mathcal{L}^d_{even}, 2d \geq h$. $\mathcal{L}_{odd} \equiv \mathcal{L}^d_{odd}, 2d + 1 \geq h$. $\mathcal{G}_{even} \equiv \mathcal{G}^d_{even}, 2d \geq h$. $\mathcal{G}_{odd} \equiv \mathcal{G}^d_{odd}, 2d + 1 \geq h$.

Les deux propriétés suivantes découlent de l'algorithme $\mathcal{DP}$.

Propriété 5.2 *Soit c une configuration telle que* $c \in \mathcal{L}_{cs} \wedge \mathcal{L}^d_{odd} \wedge \mathcal{G}^d_{even}$. *Après un cycle asynchrone d'exécution, le système atteint une configuration* $c' \in \mathcal{L}_{cs} \wedge \mathcal{L}^{d+1}_{even} \wedge \mathcal{G}^d_{odd}$.

Propriété 5.3 *Soit* $c \in \mathcal{L}_{cs} \wedge \mathcal{L}^d_{even} \wedge \mathcal{G}^{d-1}_{odd}$. *Après un cycle asynchrone d'exécution, le système atteint une configuration* $c' \in \mathcal{L}_{cs} \wedge \mathcal{L}^d_{odd} \wedge \mathcal{G}^d_{even}$.

Lemme 5.13 *Partant d'une configuration quelconque* $c \in \mathcal{C}$, *le système atteint en h ou* $h-1$ *cycles asynchrones d'exécution une configuration* $c' \in \mathcal{L}_{cs} \wedge \mathcal{L}^0_{odd}$.

Preuve: Par le lemme 5.12, en au plus $h - 1$ cycles asynchrones d'exécution, tous les processeurs sont prêts (c'est à dire que $\mathcal{L}_{cs}$ est vrai). Si tous les processeurs enfants de la racine ont la même couleur que lui, alors $\mathcal{L}^0_{odd}$ est vrai. Sinon supposons qu'il existe un nœud enfant i de la racine et tel que $Couleur_i \neq Couleur_r$. Puisque le processeur P_i est prêt, tous ses processeurs enfants ont la même couleur que lui, et la règle $\mathcal{DP} : \mathcal{R}_2$ est activable sur le processeur P_i. Au cours du cycle asynchrone d'exécution suivant, P_i en exécutant $\mathcal{DP} : \mathcal{R}_2$, copie la couleur du processeur racine, et $\mathcal{L}^0_{odd}$ devient vrai. □

Définition 5.5

$$\mathcal{L}_{oe} \equiv \mathcal{L}_{cs} \wedge ((\mathcal{L}_{odd} \wedge \mathcal{G}_{even}) \cup (\mathcal{L}_{even} \wedge \mathcal{G}_{odd}))$$

Lemme 5.14 *En partant d'une configuration* $c \in \mathcal{L}_{cs} \wedge \mathcal{L}^0_{odd}$, *le système atteint en au plus* $h - 1$ *cycles asynchrones d'exécution une configuration* $c' \in \mathcal{L}_{oe}$.

Preuve: Soit $c \in \mathcal{L}_{cs} \wedge \mathcal{L}^0_{odd} \wedge \mathcal{G}^0_{even}$. En partant de c, en un cycle asynchrone d'exécution, le système atteint une configuration $c_1 \in \mathcal{L}_{cs} \wedge \mathcal{L}^1_{even} \wedge \mathcal{G}^0_{odd}$ (par la propriété 5.2). Maintenant, en partant de la configuration c_1, le système atteint une configuration $c_2 \in \mathcal{L}_{cs} \wedge \mathcal{L}^1_{odd} \wedge \mathcal{G}^1_{even}$ (par la propriété 5.3). Par conséquent, après $h-1$ cycles asynchrones d'exécution en partant de la configuration c, le système atteint une configuration $c' \in \mathcal{L}_{oe}$. □

Les propriétés suivantes découlent du lemme 5.14 et des propriétés 5.2 et 5.3.

Propriété 5.4 *En partant d'une configuration $c \in \mathcal{L}_{cs} \wedge \mathcal{L}_{even} \wedge \mathcal{G}_{odd}$, en un cycle asynchrone d'exécution, le système atteint une configuration $c' \in \mathcal{L}_{cs} \wedge \mathcal{L}_{odd} \wedge \mathcal{G}_{even}$. En partant d'une configuration $c \in \mathcal{L}_{cs} \wedge \mathcal{L}_{odd} \wedge \mathcal{G}_{even}$, en un cycle asynchrone d'exécution, le système atteint une configuration $c' \in \mathcal{L}_{cs} \wedge \mathcal{L}_{even} \wedge \mathcal{G}_{odd}$.*

Propriété 5.5 *En partant d'une configuration $c \in \mathcal{L}_{oe}$, les processeurs à distance impaire de la racine et les processeurs à distance paire de la racine exécutent une action alternativement suivant les cycles asynchrones d'exécution.*

Théorème 6 *En partant d'une configuration $c \in \mathcal{L}_{cs} \wedge \mathcal{L}^0_{odd}$, une séquence de m messages diffusés par la racine atteint tous les processeurs en au plus $h + 2m - 1$ cycles asynchrones d'exécution.*

Preuve: Supposons que $c \in \mathcal{L}_{cs} \wedge \mathcal{L}^0_{odd}$. Le processeur racine peut émettre son premier message dans la configuration c. Il faut h cycles asynchrones d'exécution pour qu'un message émis par la racine soit reçu par tous les processeurs du réseau arborescent.

Par la propriété 5.5, la racine est en mesure d'émettre un nouveau message tous les deux cycles asynchrones d'exécution. Donc, en partant de la configuration c, la racine envoie le $m^{\text{ième}}$ message dans le $2 \times m - 1^{\text{ième}}$ cycle asynchrone d'exécution et il faut à ce dernier message h cycles asynchrones d'exécution supplémentaires pour atteindre tous les processeurs du réseau arborescent.

Par conséquent, le nombre maximum de cycles asynchrones d'exécution nécessaire à la diffusion de m messages dans l'arbre en partant de la configuration c est de $h + 2m - 1$. □

En partant d'une configuration quelconque, il faut au plus $2h+2m-1$ cycles asynchrones d'exécution pour que tous les processeurs aient reçu m messages diffusés par le processeur racine (par le lemme 5.13 et le théorème 6).

5.6 Résumé

Dans ce chapitre, nous avons montré comment synchroniser de manière auto-stabilisante des processeurs dans un réseau en arbre soit globalement (un processeur exécute ses actions au même rythme que tous les autres processeurs du système) soit localement (un processeur exécute ses actions au même rythme que ses processeurs voisins). La synchronisation globale permet de construire facilement des algorithmes auto-stabilisants à partir d'une fonction globale, quand la synchronisation locale permet d'obtenir un meilleur parallélisme des actions exécutées par les processeurs.

Les deux outils que nous proposons utilisent une mémoire constante (2 bits par processeur pour le synchroniseur global et 1 bit par processeur pour le synchroniseur local). En outre, le synchroniseur local est instantanément stabilisant : dès que les défaillances transitoires ont cessé, il exhibe immédiatement un comportement correct. Ce dernier point est particulièrement utile dans le cas de réseaux hiérarchiques évoluant dynamiquement, puisque les changement de topologie peuvent être assimilées à des défaillances transitoires.

Chapitre 6

Communications à Délai Borné

Dans cette dernière partie, nous présentons une technique générale pour réduire systématiquement le coût des communications. Plus précisément, nous développons un modèle où les messages transmis entres les processeurs ne sont pas atomiques, et de ce fait peuvent être retransmis et modifiés à la volée par les ordinateurs se trouvant sur le chemin de la communication. Cette approche fait que pour de nombreuses applications, il suffit à chaque processus d'avoir en mémoire un nombre borné (et petit) de bits.

6.1 Références

Dans le schéma classique de routage du modèle à *passage de messages*, un seul message est transmis à un instant donné sur un lien de communication. Quand un message arrive sur un processeur intermédiaire sur le chemin vers sa destination, il doit être complètement réceptionné et stocké par le processeur intermédiaire. Le message attend alors dans un tampon qu'un lien de communication sortant soit disponible et finalement, quand son tour arrive, le message est intégralement retransmis sur un autre lien.

Il y a plusieurs aspects négatifs qui apparaissent lors de l'utilisation d'un routage à passage de messages. Chaque processeur intermédiaire nécessite une mémoire locale suffisante pour stocker un message entier. La recopie de et vers les liens de communications introduisent un surcoût important. Le délai de mise des messages dans le tampon peut également être significatif.

Le routage *à délai borné*, dont nous traitons ici, a été introduit pour remédier aux défauts susmentionnés. Il est utilisé dans de nombreux réseaux industriels en anneau (par exemple Token Ring et FDDI). Dans ce schéma de routage, un processeur peut commencer à retransmettre un message au prochain processeur sur le chemin avant qu'il ait entièrement réceptionné le message. Plusieurs parties d'un message peuvent alors se trouver en transit à un instant donné sur des liens différents et des processeurs différents. En même temps que le premier bit du message est réceptionné puis transmis sur un câble sortant, le lien correspondant est réservé, et la réservation d'un lien est libérée quand le dernier bit du message a

été transmis sur le lien.

Cette approche supprime le besoin d'avoir une mémoire locale supérieure à la taille nécessaire au stockage d'un message, et un nombre borné et petit de bits s'avère suffisant. En conséquence, le délai (qui pouvait être proportionnel à la taille du message) devient borné puisqu'il dépend uniquement de la mémoire locale attribuée au tampon.

Dans la littérature, les travaux qui concernent les algorithmes auto-stabilisants dans un routage à délai borné traitent principalement de la transformation d'algorithmes à délai borné classiques en leurs équivalents auto-stabilisants. Dans [TB96], une version auto-stabilisante du protocole industriel *Token Ring* est présentée, et dans [CV99], une version auto-stabilisante du protocole *FDDI* est donnée. Nous prenons dans ce chapitre l'approche duale en présentant deux algorithmes "naturellement" auto-stabilisants et en les transformant de manière à les amener à supporter un routage à délai borné.

Les applications que nous donnons dans ce chapitre sont en fait basées sur un circuit virtuel qui est construit de manière auto-stabilisante. Plusieurs approches ont été abordées dans la littérature, parmi lesquelles [Tch81] et [AHOvdP94]. Cependant, ces solutions souffrent de plusieurs défauts :

1. Elles requièrent qu'un prétraitement soit effectué sur le système avant que le protocole de construction de circuit virtuel ne soit lancé. D'un certaine manière, ceci contredit les hypothèses de l'auto-stabilisation, puisque l'algorithme est basé sur le fait que des connaissances globales fiables ont été obtenues au préalable au lancement du programme.
2. Elles ne fonctionnent pas en modèle à délai borné, mais en modèle à passage de messages.

Contrairement à ce type de solutions, notre proposition ne présuppose aucun prétraitement, mais requiert que l'algorithme soit exécuté sur un système dont le graphe de communication est non seulement fortement connexe, mais également eulérien (chaque processeur possède autant de câbles entrants que de câbles sortants). Le prérequis d'un graphe de communication eulérien apparait naturel dans la mesure où, pour supporter un routage à délai borné, un processeur doit être capable de retransmettre l'information reçue sur chacun de ses liens entrants à au moins un de ses liens sortants.

Nous présentons tout d'abord un algorithme auto-stabilisant de construction de circuit eulérien qui satisfait aux contraintes de délai borné. Notons que les graphes eulériens comprennent tous les graphes où les communications sont bidirectionelles, qui sont parmis les plus utilisés en algorithmique répartie. Cette construction peut être utilisée pour exécuter des algorithmes auto-stabilisants fonctionnant avec un délai borné sur des anneaux unidirectionnels, comme [TB96] ou [CV99], sur des graphes eulériens. Nous présentons les problèmes ayant trait aux contraintes du délai borné dans le contexte particulier de l'auto-stabilisation, et mettant en avant une technique de propagation de marqueur afin de réinitialiser les messages en cours de retransmission.

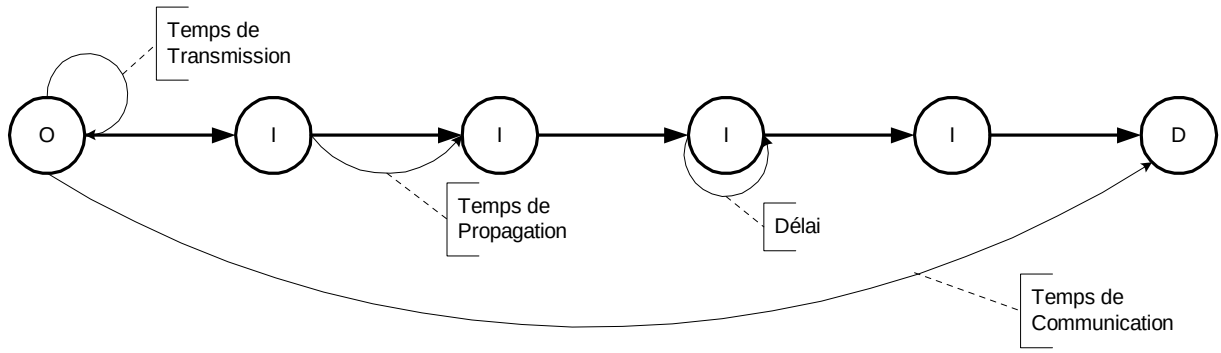

FIG. 6.1 – Le temps de communication entre O et D.

En application, nous présentons tout d'abord un algorithme d'élection, basé sur les identifiants des processeurs du réseau, puis un algorithme de recensement, qui présente la particularité de ne pas stocker les informations (de taille importante) sur les processeurs mais dans les messages en transit dans le réseau. A la différence de toutes les solutions précédentes, nos applications supportent les contraintes du routage à délai borné.

6.2 Hypothèses Spécifiques au Chapitre

6.2.1 Temps de Communication

Nous définissons plusieurs paramètres afin de calculer le temps de communication sous un routage à délai borné. La plupart de ces définitions sont extraites de [Wal91].

Le *temps de transmission* T_{t_i} sur le processeur P_i est le temps nécessaire au processeur P_i pour recevoir (envoyer) un message depuis (vers) un lien de communication. Par exemple, une carte réseau à $100\,Mbs^{-1}$ transmet un message de $5\,Mb$ en $0,05\,s$. Le temps de transmission est donné par la relation :

Définition 6.1 (Temps de Transmission) *Le* temps de transmission *sur le processeur P_i est $T_{t_i} = \frac{l}{C}$, où l est la taille du message en nombre de bits et C est de débit du lien de communication.*

Le *temps de propagation* $T_{p_{i\to j}}$ du processeur P_i au processeur P_j est le temps nécessaire à un bit pour traverser un lien de communication en deux processeurs P_i et P_j. Par exemple, la célérité de l'unité d'information sur une paire torsadée utilisée dans les réseaux Ethernet est d'environ $200\,000\,Km.s^{-1}$, donc une unité d'information traverse un câble de $20\,m$ en $0,0000001\,s$. Le temps de propagation est donné par la relation :

Définition 6.2 (Temps de Propagation) *Le* temps de propagation *du processeur P_i au processeur P_j est $T_{p_{i\to j}} = \frac{d}{\tau}$, où d est la longueur du lien et τ est la célérité de l'unité d'information dans le milieu constitué par le lien.*

Le *délai* T_{d_i} sur le processeur P_i est le temps nécessaire au remplissage du tampon d'un processeur P_i et est donné par la relation :

Définition 6.3 (Délai) *Le* délai *sur le processeur P_i est $T_{d_i} \leq \frac{b}{C}$, où b est la taille du tampon en nombre de bits, et C le débit de transmission du lien de communication.*

Nous sommes maintenant en mesure d'exprimer le temps de communication entre deux processeurs P_i et P_j qui ne sont pas des voisins l'un de l'autre. Le *temps de communication* $T_{c_{i\to j}}$ entre les processeurs P_i et P_j est temps nécessaire à un message pour aller de P_i à P_j en suivant le chemin i, i_1, i_2,■, i_k, j dans le graphe de communication. Ce temps est donné par la relation :

Définition 6.4 (Temps de Communication) *Le temps de communication du processeur P_{i_0} au processeur P_{i_k} est*

$$\begin{aligned} T_{c_{i_0\to i_k}} &= T_{t_{i_0}} + T_{p_{i_0\to i_1}} + T_{d_{i_1}} + \cdots + T_{p_{i_{k-2}\to i_{k-1}}} + T_{d_{i_{k-1}}} + T_{p_{i_{k-1}\to i_k}} \\ &= T_{t_{i_0}} + \sum_{l=0}^{k-2} \left(T_{p_{i_l\to i_{l+1}}} + T_{d_{i_{l+1}}} \right) + T_{p_{i_{k-1}\to i_k}} \end{aligned}$$

6.2.2 Modèle de Communication

Le modèle que nous utilisons dans ce chapitre est un système réparti où les processeurs sont liés entre eux par des câbles. Pour rendre compte du fait que les messages ne sont plus atomiques, nous les décomposons en éléments.

Messages

Un message (ou *signal*) est composé d'*éléments de signal*. Un élément de signal est de l'un des types suivants :

1. élément de signal **nul**, dénoté par `N`. L'élément de signal `N` fait office de séparateur de message pour permettre que plusieurs messages soient présents dans le réseau à un instant donné.
2. élément de signal **binaire**, dénoté par `0` ou `1`. Les éléments de signal `0` et `1` sont les valeurs booléennes servant à coder les variables contenues dans les messages.
3. élément de signal **marqueur**, dénoté par `Z`. L'élément de signal `Z` est un élément spécial qui permet la réinitialisation des messages par un algorithme.

Nous définissons un type abstrait énuméré (`N`, `Z`, `0`, `1`), pour les éléments de signal. Dans le modèle *à délai borné*, les messages sont des séquences de éléments de signal contenant des `0`s, des `1`s et des `Z`s entourés d'éléments de signal `N` ; c'est à dire que les messages sont exprimés sous la forme d'une expression régulière : $\mathtt{N}^\omega(\mathtt{1}|\mathtt{0}|\mathtt{Z})^\omega\mathtt{N}^\omega$. Les éléments de signal `N` servant uniquement à délimiter les messages dans le réseau, ceux-ci ne font pas partie du contenu du message.

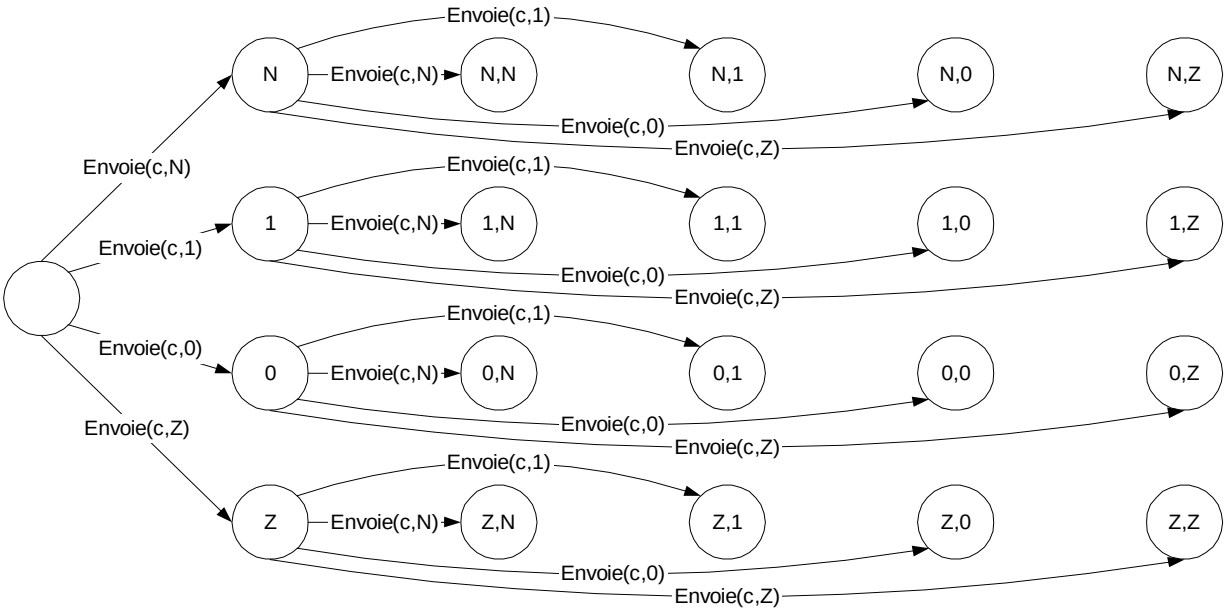

FIG. 6.2 – Un câble FIFO de capacité 2 dans le modèle à délai borné.

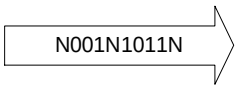

FIG. 6.3 – Un câble de capacité 10.

Dans la suite du chapitre, un message en transit est surmonté d'une flèche indiquant quels sont les éléments de signal devant être reçus en premier lors de la réception de ce message par un processeur. Par exemple, le message `111100` dont les deux `0` sont les premiers à être reçus est noté $\overrightarrow{\texttt{111100}}$.

Câbles

Les câbles que nous considérons dans ce chapitre sont unidirectionnels et de taille fixée (voir chapitre 2). Leur taille (ou *capacité*) est exprimée en nombre de éléments de signal. Par ailleurs, les éléments de signal contenus dans ces câbles sont ordonnés et les actions *Envoie* et *Reçoît* sont ordonnées de telle manière que les ordres d'émission et de réception des messages sont respectés (les câbles sont FIFO).

La figure 6.2 présente une partie d'un automate représentant un câble FIFO de capacité 2 et pouvant contenir des éléments de signal `N`, `1`, `0` ou `Z`. Pour une plus grande lisibilité de la figure, seules les actions d'écriture (*Envoie*) ont été représentées. Les actions de lectures peuvent être ajoutées suivant les règles définies au chapitre 2.

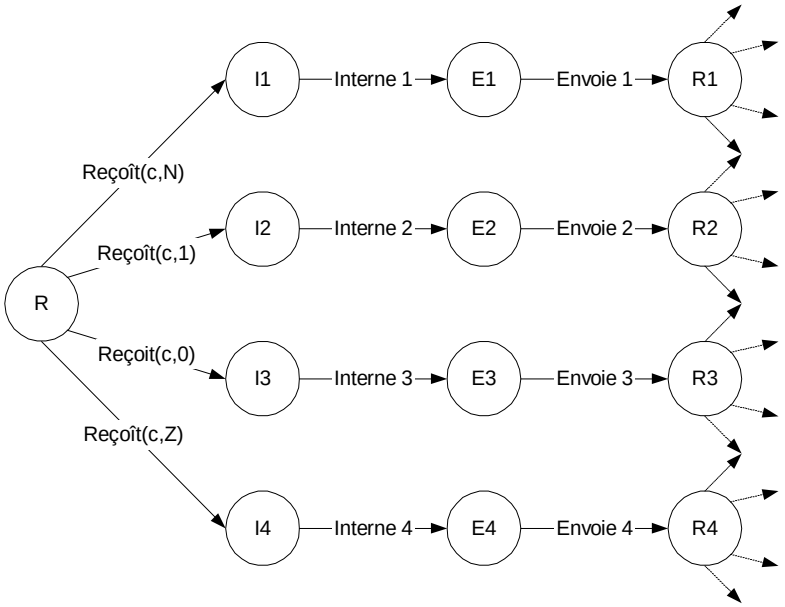

FIG. 6.4 – Un processeur dans le modèle à délai borné.

Dans la suite, les câbles sont caractérisés uniquement par leur capacité c. Leur contenu est alors une séquence d'au plus c éléments de signal. La figure 6.3 présente un câble de capacité 10 et contenant deux messages : $\overrightarrow{\texttt{001}}$ et $\overrightarrow{\texttt{1011}}$.

Processeurs

Les processeurs que nous considérons dans ce chapitre fonctionnent tous suivant le même schéma : ils effectuent de manière répétitive une action de lecture sur chacun de leurs câbles entrants, puis une séquence finie d'actions internes, enfin une action d'écriture sur chacun de leurs câble sortants.

Il est notable qu'un processeur en phase de lecture doit être capable, sous peine de blocage, de lire chacun des éléments de signal possibles (à savoir `N`, `1`, `0` ou `Z`). Sur la figure 6.4, un automate partiel représentant un processeur doté d'un unique câble entrant est représenté.

Pour une meilleure compréhension des algorithmes locaux exécutés par les processeurs, nous les décrivons sous la forme d'une fonction activée sur *réception d'un message*. La réception d'un message est simplement la lecture d'un élément de signal différent de `N` après la lecture d'un élément de signal `N`. Afin d'exprimer le délai à chaque processeur, nous utilisons des *tampons* situés sur chacun des processeurs.

Un *tampon* est une structure de données qui contient des éléments de signal. En l'occurrence, le *tampon* que nous considérons est une structure dotée de possibilités d'accès aléatoire. Il est possible de manipuler ce *tampon* au moyen des fonctions suivantes :

- `Ajoute(b,s)` : ajoute l'élément de signal `s` en queue du *tampon* `b`. Si `b` est plein, l'élément de queue est simplement remplacé par `s`.
- `Supprime(b)` : supprime l'élément de signal placé en tête du *tampon* `b`. Si `b` est vide, la fonction n'a aucun effet.
- `Lit(b,index)` : retourne l'élément de signal placé à la position `index` dans le *tampon* `b`. L'argument `index` est compris entre `0` (tête) et `Taille(b) - 1` (queue). Si `index` est en dehors de ces limites, `Lit` retourne `N`.
- `Ecrit(b,index,s)` : écrit l'élément de signal `s` à la position `index` dans le *tampon* `b`. L'argument `index` est compris entre `0` (tête) et `Size(b) - 1` (queue). Si `index` est en dehors de ces limites, `Ecrit` n'a aucun effet.
- `LectureElement(b)` : lit un élément de signal `s` depuis le lien de communication entrant. Si `s` est `0` ou `1`, la fonction `Ajoute(b,s)` est appelée et *vrai* est renvoyé. Si l'élément de signal lu n'est ni `0` ni `1`, la fonction renvoie *faux*.
- `EcritureElement(b)` : écrit un élément de signal vers un lien de communication sortant. Si le *tampon* `b` est vide, un `N` est écrit sur le câble sortant et *faux* est retourné. Dans le cas contraire, `Lit(b,0)` est écrit sur le câble de sortie, puis la fonction `Supprime(b)` est appelée, et finalement `EcritureElement` retourne *vrai*.
- `Taille(b)` : renvoie la taille courante du *tampon* `b`, c'est à dire le nombre d'éléments de signal présents dans le *tampon*. Notons que l'invocation des fonctions `Ajoute`, `Supprime`, `LectureElement`, et `EcritureElement` sur un *tampon* peuvent changer la taille de ce *tampon*, mais que les fonctions `Lit` et `Ecrit` n'ont aucun effet sur la taille du *tampon*.
- `Efface(b)` : efface le *tampon* `b` en appelant répétitivement `Supprime(b)` tant que `Taille(b)` est supérieur à 0.

Systèmes

Les systèmes répartis que nous considérons dans ce chapitre sont construits à partir des processeurs et des câbles suivant l'approche décrite au chapitre 2. Pour faciliter l'écriture des preuves de la section 6.3, nous considérons que les exécutions du système s'effectuent sous le démon *synchrone*. Cette hypothèse est justifiée par le modèle physique que nous avons adopté, et qui suppose qu'un temps global absolu existe. Notre hypothèse revient à supposer que tous les processeurs sont matériellement identiques (aux capacités de communication près).

Exemple La figure 6.5 présente un système réparti où les câbles sont de capacité 10 et où les processeurs ont un tampon de taille 2. Il y a 5 messages en transit sur ce système : $\overrightarrow{\texttt{00001000Z}}$, $\overrightarrow{\texttt{101}}$, $\overrightarrow{\texttt{00}}$, $\overrightarrow{\texttt{Z1}}$, $\overrightarrow{\texttt{00100}}$ et $\overrightarrow{\texttt{001}}$.

Les Contraintes du Délai Borné

Un algorithme à délai borné doit satisfaire certaines caractéristiques :

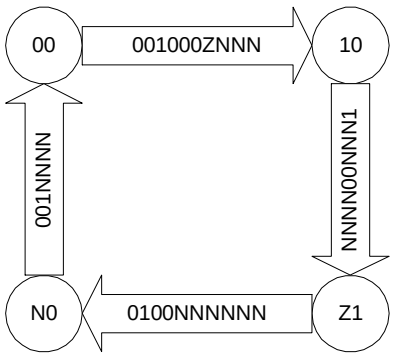

FIG. 6.5 – Un système réparti dans le modèle à délai borné.

1. Contrainte de **Débit :** les actions `EcritureElement` et `LectureElement` doivent être exécutées en alternance stricte ;
2. Contrainte de **Délai** : la taille du tampon de chaque processeur est constante.

Routage à Délai Borné dans les Graphes Eulériens

Si le graphe de communication n'est pas un anneau unidirectionnel, alors au moins un processeur possède deux câbles entrants. De manière à prendre en compte l'arrivée simultanée de plusieurs messages, nous requérons que :

1. Les processeurs possèdent autant de buffers que de câbles entrants ;
2. Les processeurs possèdent autant de câbles sortants que de câbles entrants. Notons que dans tout graphe, le nombre total de câbles entrants est égal au nombre total de câbles sortants. Par conséquent, imposer que sur chaque processeur il n'y ait pas plus de câbles entrants que sortants implique que sur chaque processeur, ces deux nombres sont égaux.

Exemple La figure 6.6 présente un système réparti dans le modèle à délai borné. Un des processeurs est représenté par un rond doté d'un bord gras : il possède plus d'un câble entrant. Il possède aussi deux buffers, l'un contenant `10` et l'autre `Z0`. Cependant, le processeur peut choisir quel tampon est associé à quel câble, de manière à ce que l'une des deux options suivantes soit activée :

1. le câble entrant à l'ouest est lié au câble sortant au sud au moyen du tampon supérieur, et le câble entrant à l'est est lié au câble sortant au sud-est au moyen du tampon inférieur ;
2. le câble entrant à l'est est lié au câble sortant au sud au moyen du tampon supérieur, et le câble entrant à l'ouest est lié au câble sortant au sud-est au moyen du tampon inférieur.

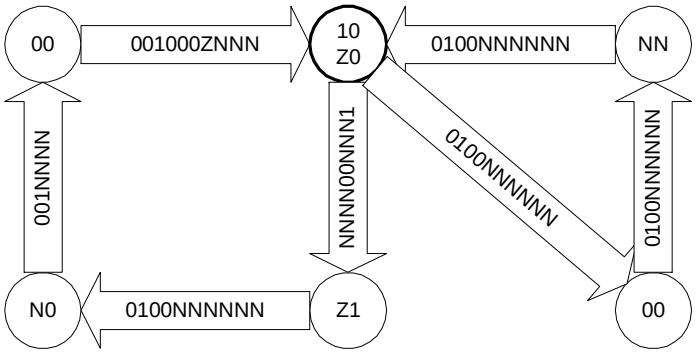

FIG. 6.6 – Un système réparti dans le modèle à délai borné dont le graphe de communication est Eulérien.

6.2.3 Opérations de Base

Avec la contrainte d'un délai de retransmission borné, même les tâches algorithmiques les plus simples peuvent poser problème. En effet, un processeur est capable d'effectuer n'importe quelle modification sur un message, mais seulement en utilisant une connaissance limitée de son contenu. Une partie de cette connaissance est la position courante dans le message en cours de retransmission.

Comme chaque élément de signal entrant peut impliquer un comportement différent de l'algorithme local du processeur, nous pouvons supposer que cette position se trouve "cachée" dans le compteur de programme, et n'implique pas de surcoût. Les paragraphes suivants décrivent les opérations de base utilisées par les algorithmes présentés dans la section 6.4, en mettant en évidence que certaines de ces opérations peuvent être plus ou moins facilement implantées selon la manière dont les données sont représentées.

Codage des Entiers

Les applications réparties que nous présentons utilisent des variables entières. Une manière classique de représenter de telles variables est le codage binaire canonique : chaque nombre est simplement écrit en base 2. Comme l'indiquent les paragraphes suivants, la manière dont sont ordonnés les bits influe sur la difficulté à implanter certaines opérations.

Comparaison

Le but de cette opération est d'effectuer une comparaison entre une valeur contenue dans un message et une valeur (variable ou constante) contenue dans un processeur. De plus, afin de garantir un délai borné sur chaque processeur, les variables sont supposées potentiellement plus grandes que le tampon du processeur utilisé pour retransmettre le message (c'est à dire que le nombre de bits nécessaire au codage de la variable est strictement supérieur à la taille du tampon du processeur).

Une comparaison est facilement effectuée en utilisant le codage binaire canonique, pourvu que les bits de poids fort soient les premiers à apparaître : le premier bit différent entre les deux valeurs indique laquelle des deux valeurs est la plus grande, et l'abscence d'un tel bit indique que les deux valeurs sont égales.

Si les bits de poids fort viennent en dernier, il est seulement possible de tester la différence à la volée, suivant qu'il existe ou non un bit de différence entre les deux valeurs.

Remplacement

Le but de cette opération est de remplacer une valeur à la volée (avec un tampon de taille constante) : soit remplacer une variable locale par une variable contenue dans un message, soit remplacer une variable contenue dans un message par une variable ou une constante locale. Cette opération peut être effectuée à la volée quel que soit l'ordonnancement utilisé pour les poids des bits constituants la variable.

Réinitialisation

Le but de cette opération est de réinitialiser un message quand une erreur a été détectée relativement au contenu du message. Or, comme le tampon de chaque processeur est borné, le début du message a déjà pu être retransmis sur un canal sortant. Par exemple, dans les applications présentées dans la section 6.4, il est parfois indispensable de réinitialiser tout ou partie d'un message après en avoir pris connaissance.

Une telle opération ne peut être effectuée en une seule étape, puisque le message à modifier a déjà transité par le processeur lorsque l'ordre de réinitialisation intervient. Nous utilisons alors une technique de marqueur pour effectuer cette opération : un symbole spécial (l'élément de signal `Z`) est ajouté en queue du message à réinitialiser. Chacun des processeurs suivants sur le trajet du message déplace ce marqueur autant qu'il est en est capable, tout en réinitialisant les bits concernés (en les remplaçant par des éléments de signal `N`). Quand le marqueur `Z` arrive au début du message, un nouveau message (dépendant de l'application) peut alors être retransmis. Plusieurs marqueurs `Z` peuvent être présents dans un message au même instant. Après que le premier de ces marqueurs est arrivé au début du message, tous les autres sont simplement ignorés.

Exemple Considérons un message contenant un marqueur `Z` et des processeurs possédant des Buffers de taille 3. A chaque traversée d'un nouveau processeur, le marqueur est avancé de 2 positions.

Message Entrant	tampon du Processeur	Message Sortant	Commentaires
NNZ0101	[NN]		
NNZ010	[1NN]		LectureElement
NNZ010	[1N]	N	EcritureElement
NNZ01	[01N]	N	LectureElement
NNZ01	[01]	NN	EcritureElement
NNZ0	[101]	NN	LectureElement
NNZ0	[10]	1 NN	EcritureElement
NN Z	[010]	1 NN	LectureElement
NN Z	[01]	01NN	EcritureElement
NN	[Z01]	01NN	LectureElement
NN	[01Z]	01NN	Avancement du marqueur
NN	[01]	Z01NN	EcritureElement
N	[N01]	Z01NN	LectureElement
N	[N0]	1Z01NN	EcritureElement
	[NN0]	1Z01NN	LectureElement
	[NN]	01Z01NN	EcritureElement

Fatalement, un processeur P_i finit par trouver Z au début du message. Alors P_i est en mesure de ne pas tenir compte de ce message entrant et d'initier un nouveau message. Contrairement à ce qui se passe avec un routage store-and-forward, la réinitialisation effective du message ne se produit pas immédiatement, mais peut être retardée suivant la taille de chaque tampon disponible sur les différents processeurs du système.

Si un processeur reçoît une séquence qui commence par un marqueur Z, comme dans 0010010011Z, toute donnée située après Z est ignorée. Dans l'exemple suivant, tout le message est ignoré, et le nouveau message 00 est envoyé :

Message Entrant	tampon du Processeur	Message Sortant	Commentaires
0010010011Z	[]		
0010010011	[Z]		LectureElement
0010010011	[Z]	0	EcritureElement
001001001	[Z]	0	LectureElement
001001001	[Z]	00	EcritureElement
00100100	[Z]	00	LectureElement
00100100	[Z]	N00	EcritureElement
⋮	⋮	⋮	
	[]	NNNNNNNNN00	Le reste du message est ignoré

Il ressort des paragraphes précédents que les modifications faites sur les messages par les processeurs peuvent être réparties en deux catégories distinctes : *(i)* l'opération peut être effectuée à la volée, et aucun problème de délai ne survient, et *(ii)* l'opération implique la modification de données déjà retransmises, et un mécanisme de marqueur est utilisé.

6.3 Construction d'un Circuit Eulérien

Les applications que nous présentons dans la section 6.4 sont supposées être exécutées sur un réseau en anneau unidirectionnel. Dans cette partie, nous montrons comment construire un circuit virtuel où circule un ou plusieurs messages et où chaque processeur effectue une action de niveau supérieur (c'est à dire une action de l'application) exactement une fois à chaque parcours par un message du circuit virtuel.

Les protocoles sont dirigés par l'arrivée de messages, donc si aucun message n'est présent à l'origine dans le réseau, il y a un blocage. Nous supposons donc qu'il existe un temporisateur sur l'un des processeurs qui permet de résoudre une telle situation (c'est à dire qu'après avoir observé pendant une longue période qu'aucun message n'est apparu, il envoie un nouveau message sur l'un de ses câbles sortants). De toute façon, cette caractéristique consommatrice de temps ne peut être utilisée qu'une seule fois, et uniquement en cas de mauvaise initialisation (initialisation sans message).

Le schéma de construction d'une application auto-stabilisante sous les contraintes à délai borné se fait par composition de plusieurs algorithmes :

1. Un premier algorithme (voir section 6.3.1) s'assure qu'à partir de n'importe quelle configuration initiale, il reste un unique message dans tout le réseau au bout d'un temps fini ;
2. Un second algorithme (voir section 6.3.2, algorithme 5) construit un circuit Eulérien sur le réseau, pourvu qu'un unique message soit présent ;
3. Le dernier algorithme (le protocole lié à l'application et supposé s'exécuter sur un anneau unidirectionnel) implante une solution à la spécification du problème : une élection de chef dans la section 6.4.1, un recensement dans la section 6.4.2.

6.3.1 Suppression des Messages

Le but de ce protocole est de supprimer les messages résultant d'une mauvaise initialisation de telle sorte qu'au bout d'un temps fini, il ne subsiste plus qu'un unique message.

Description Informelle

Chaque message arrivant sur un processeur est retardé avec probabilité $\frac{1}{2}$, donc les messages avancent à des vitesses différentes. Le protocole de suppression des messages réduit alors le nombre de messages de deux manières :

1. En *concaténant* deux messages qui se poursuivent sur un même trajet ;
2. En *détruisant* un ou plusieurs messages arrivant *simultanément* sur un processeur.

Retard

Chaque processeur du système, lors de la réception d'un message (c'est à dire du passage de l'élément de signal `N` à un autre élément de signal) décide de "retarder" le message entrant (en utilisant une case spécifique de son *tampon*) avec une probabilité $\frac{1}{2}$. De la sorte, si plusieurs messages sont initialement présents dans le réseau, alors ils avancent à des vitesses différentes. Si deux messages sont retransmis répétitivement sur le même cycle, ils finissent par se rattraper, et ils sont alors concaténés.

Exemple de Retard Considérons le système où deux messages $\overrightarrow{\texttt{011}}$ et $\overrightarrow{\texttt{001}}$ séparés par deux éléments de signal `N` se trouvent dans le réseau à la suite l'un de l'autre. Ils arrivent sur le même processeur, qui lance une pièce à l'arrivée de chaque message afin de déterminer s'il le retarde ou non. Admettons que les lancers de pièces soient tels que le processeur décide de retarder le premier message et pas le deuxième. Le retard est créé artificiellement en lisant l'élément de signal suivant du message et en envoyant sur le câble de sortie un élément `N`. En fait, ceci revient à placer immédiatement *avant* le message l'élément de signal `N` qui est placé immédiatement *après* lui.

Evidemment, dès la fin de la réception du premier message (retour à la réception d'un élément de signal `N`), le premier `N` suivant le message est ignoré : ceci peut être fait sans rompre la conformité au routage à délai borné puisque le *tampon* du processeur dispose d'un élément de signal supplémentaire. Le deuxième message, lui, n'est pas retardé, ce qui implique qu'à la sortie du processeur, l'espace entre les deux messages (qui était de deux éléments de signal `N`) a strictement diminué (il n'est plus que d'un élément de signal `N`). Ceci se trouve résumé sur le tableau suivant :

Messages Entrant	tampon du Processeur	Messages Sortant	Commentaires
$\overrightarrow{\texttt{001}}\overrightarrow{\texttt{NN011}}$	[N]		
$\overrightarrow{\texttt{001}}\texttt{NN}\overrightarrow{\texttt{01}}$	[1N]		LectureElement
$\overrightarrow{\texttt{001}}\texttt{NN}\overrightarrow{\texttt{01}}$	[1NN]		Lancer vrai
$\overrightarrow{\texttt{001}}\texttt{NN}\overrightarrow{\texttt{01}}$	[1N]	N	EcritureElement
$\overrightarrow{\texttt{001}}\texttt{NN}\,\overrightarrow{\texttt{0}}$	[11N]	N	LectureElement
$\overrightarrow{\texttt{001}}\texttt{NN}\,\overrightarrow{\texttt{0}}$	[11]	NN	EcritureElement
$\overrightarrow{\texttt{001}}\texttt{NN}$	[011]	NN	LectureElement
$\overrightarrow{\texttt{001}}\texttt{NN}$	[01]	$\overrightarrow{\texttt{1}}\texttt{NN}$	EcritureElement
$\overrightarrow{\texttt{001}}\texttt{N}$	[N01]	$\overrightarrow{\texttt{1}}\texttt{NN}$	LectureElement
$\overrightarrow{\texttt{001}}\texttt{N}$	[01]	$\overrightarrow{\texttt{1}}\texttt{NN}$	Réduction
$\overrightarrow{\texttt{001}}\texttt{N}$	[0]	$\overrightarrow{\texttt{11}}\texttt{NN}$	EcritureElement
$\overrightarrow{\texttt{001}}$	[N0]	$\overrightarrow{\texttt{11}}\texttt{NN}$	LectureElement
$\overrightarrow{\texttt{001}}$	[N]	$\overrightarrow{\texttt{011}}\texttt{NN}$	EcritureElement
$\overrightarrow{\texttt{00}}$	[1N]	$\overrightarrow{\texttt{011}}\texttt{NN}$	LectureElement
$\overrightarrow{\texttt{00}}$	[1N]	$\overrightarrow{\texttt{011}}\texttt{NN}$	Lancer faux
$\overrightarrow{\texttt{00}}$	[1]	$\texttt{N}\overrightarrow{\texttt{011}}\texttt{NN}$	EcritureElement
$\overrightarrow{\texttt{0}}$	[01]	$\texttt{N}\overrightarrow{\texttt{011}}\texttt{NN}$	LectureElement
$\overrightarrow{\texttt{0}}$	[0]	$\overrightarrow{\texttt{1}}\texttt{N}\overrightarrow{\texttt{011}}\texttt{NN}$	EcritureElement

Destruction

Il reste maintenant à gérer le cas des messages qui arrivent simultanément sur un processeur. Puisqu'un processeur connait la position dans le message couramment relayé, une arrivée simultanée est définie comme suit :

Définition 6.5 (Arrivée l-simultanée) *Un processeur avec k câbles entrants ($k > 1$) a une arrivée l-*simultanée *($1 < l \leq k$) si et seulement si les propriétés suivantes sont vérifiées :*
- *l messages arrivent couramment de l câbles entrants différents ;*
- *aucun des messages entrants n'a encore été retransmis.*

Ainsi, chaque processeur dont le degré entrant est au moins 2 teste s'il reçoit au moins deux têtes des deux messages simultanément (au sens de la définition 6.5). Si ce test est vérifié, tous les messages dont l'arrivée est simultanée sont éliminés sauf un.

Dans le cas où deux messages sont retransmis répétitivement sur deux cycles qui possèdent un processeur en commun, alors il est possible qu'ils arrivent simultanément sur ce processeur, qui élimine alors l'un d'entre eux. Dans toutes les topologies eulériennes qui ne sont pas des anneaux unidirectionnels, des messages peuvent être supprimés, car il existe toujours au moins un processeur avec deux câbles entrants.

Intuition de Preuve de Correction

Puisque le contenu des messages n'a pas de signification pour l'algorithme, nous considérons simplement la tête de chacun des messages (le premier élément de signal qui ne soit pas N), en supposant que la queue du message est retransmise correctement.

Si le graphe de communication est un anneau unidirectionnel, alors les messages se déplacent à différentes vitesses de manière probabiliste. Par conséquent, il existe une probabilité non nulle pour que deux messages se rejoignent. Or, si deux messages se rejoignent, ils ne peuvent plus se séparer, car aucune partie de l'algorithme n'introduit d'élément de signal `N` à l'intérieur d'un message. Donc avec probabilité 1, il ne reste plus qu'un seul message après un temps fini.

Les graphes eulériens que nous considérons peuvent être vus comme la superposition de plusieurs anneaux unidirectionnels qui ont des processeurs en commun. Alors deux cas peuvent se produire compte tenu de l'algorithme de routage que nous utilisons (qui est en fait une exploration rotative, voir l'algorithme 5) :

1. Tous les messages suivent le même circuit qui passe par tous les arcs du graphe eulérien. Alors par un argument similaire à celui des anneaux unidirectionnels, un seul message reste au bout d'un temps fini avec probabilité 1.
2. Au moins deux messages suivent un circuit différent, mais qui présente un processeur commun P_i. Alors du fait de la probabilisation du retardement, il est possible que ces deux messages arrivent l-simultanément sur le processeur P_i, qui choisit l'un de ces messages comme étant celui à supprimer.

Dans tous les cas, le nombre de messages décroît et on peut en conclure qu'avec probabilité 1, au bout d'un temps fini, un seul message reste dans le réseau.

A partir d'un raisonnement identique à celui détaillé dans [Del95] (chapitre 5), nous en concluons que le système converge vers une configuration où il n'existe qu'un seul message en transit (de contenu arbitraire). Comme l'algorithme de destruction de message ne peut détruire de message que si l'un deux d'entre eux arrivent simultanément sur un processeur, l'unique message persiste à jamais.

Définition 6.6 (Configuration Singulière) *Une configuration c est* singulière *si et seulement si pour toute exécution $e \in \mathcal{E}_c$, exactement une tête de message m est présente dans le réseau.*

Dans la suite, nous ne considérons plus que des configurations singulières.

6.3.2 Circuit Eulérien

Le but de cet algorithme est de faire en sorte que chaque message effectue un circuit Eulérien au bout d'un temps fini pourvu qu'un unique message soit présent dans le réseau.

Algorithme

Chaque processeur P_i possède possède $\delta^-(i)$ câbles entrants et $\delta^+(i)$ câbles sortants. Comme le graphe est supposé Eulérien, nous posons $d_i = \delta^-(i) = \delta^+(i)$. En outre, chaque processeur P_i possède une variable $Chemin_i$, qui prend ses valeurs entre 0 et $d_i - 1$. Toutes les opérations d'incrément sur cette variable sont supposées être effectuées modulo d_i.

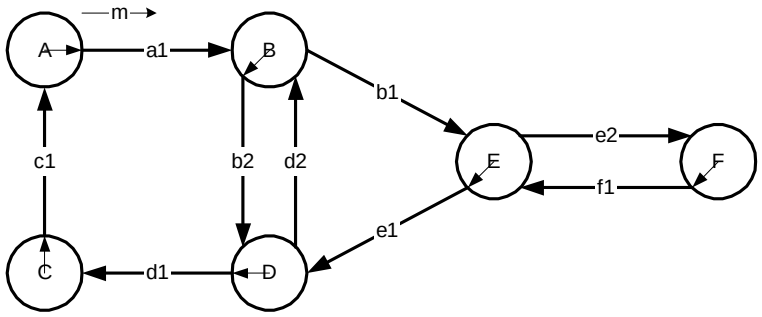

FIG. 6.7 – Le Système au Début du Premier Tour.

L'algorithme EC, exécuté sur réception de message, est le même pour tous les processeurs du système.

Algorithme 5 (Circuit Eulérien) *Algorithme du processeur P_i sur réception d'un message m*

Retransmettre m sur le câble sortant d'indice $Chemin_i$
$Chemin_i \leftarrow Chemin_i + 1$

Exemple d'Exécution La figure 6.7 présente un système dont le graphe de communication est eulérien : les processeurs A, C, E et F ont chacun un câble entrant et un câble sortant ; les processeurs B, D et E ont chacun deux câbles entrants et deux câbles sortants. La variable $Chemin_i$ est représentée pour chaque processeur P_i par une flèche qui pointe sur le câble sortant par lequel le prochain message entrant va être retransmis. Par exemple, sur la figure, le processeur A vient d'envoyer le message m et le processeur B (qui est sur le point de recevoir m) va le retransmettre sur son câble sortant b_2.

Nous examinons dans cet exemple la trajectoire du message m depuis le processeur "initiateur" A (en fait le dernier processeur à avoir retransmis le message. Etant donnée la configuration des variables $Chemin_i$, le message traverse les câbles a_1, b_2, d_1 puis c_1 avant de se retrouver de nouveau retransmis par le processeur A. Visiblement, le chemin parcouru n'est pas eulérien, puisque les chemins b_1, d_2, e_1, e_2 et f_1 n'ont pas été visités par m. Cependant, les variables $Chemin_B$ et $Chemin_D$ ont changé de valeur pendant ce premier tour de m : $Chemin_B$ pointe maintenant sur b_1 et $Chemin_D$ sur d_2.

La figure 6.8 présente le même système que celui de la figure 6.7, mais au deuxième tour du message m. Etant donné la configuration des variables $Chemin_i$, le message parcourt les câbles a_1, b_1, e_1, d_2, b_2, d_1 puis c_1 avant de se retrouver de nouveau retransmis par le processeur A. Là encore, le chemin parcouru n'est pas eulérien, puisque les chemins e_2 et f_1 n'ont pas été visités par m. Cependant, la variable $Chemin_E$ a changé de valeur durant

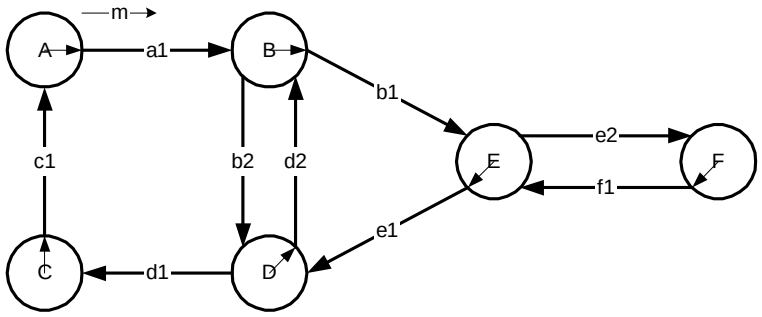

FIG. 6.8 – Le système au Début du Deuxième Tour.

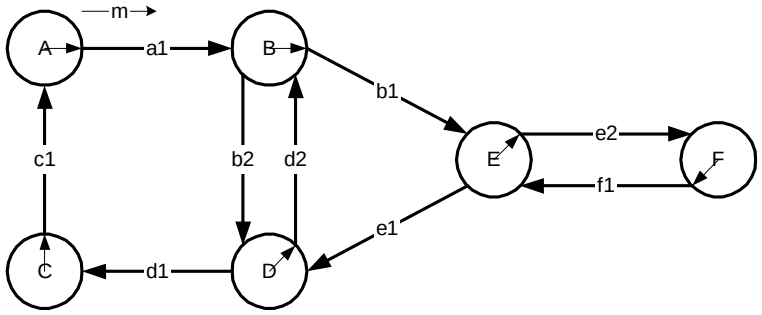

FIG. 6.9 – Le Système au Début du Troisième Tour.

ce deuxième tour de m : elle pointe maintenant sur e_2. Au contraire, les variables $Chemin_B$ et $Chemin_D$ sont revenues aux mêmes valeurs qu'au début du deuxième tour (soit respectivement b_1 et d_2), ce qui laisse présager que les processeurs correspondants sont maintenant correctement configurés pour assurer au message m qu'il effectue un parcours eulérien.

La figure 6.9 présente encore le même système au début du troisième tour. Etant donné la configuration des variables $Chemin_i$, le message parcourt les câbles a_1, b_1, e_2, f_1, e_1, d_2, b_2, d_1 puis c_1 avant de se retrouver de nouveau retransmis par le processeur A. Le chemin parcouru est bien un circuit eulérien, puisque tous les câbles ont été visités exactement une fois, et que le message m est de retour sur le processeur "initiateur" A. De plus, les variables $Chemin_i$ ont repris les mêmes valeurs qu'au début du troisième tour, ce qui signifie que le quatrième tour sera identique au troisième. En conséquence, le message m parcours indéfiniment le circuit eulérien ainsi construit.

Preuve de Correction Nous souhaitons prouver que l'algorithme 5 est bien un algorithme auto-stabilisant qui construit un circuit eulérien. Chacun des lemmes qui suivent fait hypothèse d'une configuration singulière (voir définition 6.6).

Le lemme de vivacité6.1 montre qu'indépendemment de leur ordre ou de leur fréquence

relative, toutes les actions d'émission via un câble apparaissent infiniment souvent dans toute exécution. Ceci permet aux autres lemmes de toujours considérer des facteurs d'exécution commencant et finissant par la même action d'émission. Ainsi, le lemme d'unicité 6.2 montre qu'entre deux actions successives d'émission *via* un même câble, aucun câble ne peut effectuer deux émissions. Le lemme de totalité 6.3 montre qu'après que tout câble a retransmis une fois la tête du message, entre deux actions successives d'émission sur un même câble, tout autre câble retransmet exactement une fois le message. Enfin le lemme de légitimité 6.4 prouve que ces retransmissions ont toujours lieu dans le même ordre et donc que la tête du message fini par décrire un circuit eulérien toujours identique.

Lemme 6.1 (Vivacité) *En partant d'une configuration singulière, tout processeur et tout câble est visité infiniment souvent par la tête du message.*

Preuve: Supposons qu'un lien n'est pas infiniment souvent visité par la tête du message lors d'une exécution dont le configuration initiale est singulière et ne contenant que des configurations singulières. Pour la simplicité de la preuve nous considérons qu'il s'agit du câble $c_{i\to j}$ permettant la transmission d'informations du processeur P_i vers le processeur P_j. D'après l'algorithme 5, si le processeur P_i retransmet infiniment souvent la tête du message, il le fait nécessairement sur tous ses câbles sortants. Si P_i n'a pas retransmis infiniment souvent sur le câble $c_{i\to j}$, cela signifie que le processeur P_i a retransmis seulement un nombre fini de fois la tête de message au cours de l'éxécution. Or si P_i n'a retransmis qu'un nombre fini de fois la tête de message au cours de l'éxécution, cela implique qu'il ne l'a reçu qu'un nombre fini de fois. Tous les câbles entrants de P_i sont donc dans le même cas que $c_{i\to j}$, et n'ont pas été visités infiniment souvent. En itérant le même raisonnement et comme le réseau possède un nombre fini de processeurs et de liens, on prouve qu'aucun câble et aucun processeur n'a transmis infiniment le message.

Ceci est en contradiction soit avec l'exécution de l'algorithme 5 (qui est obligatoirement infinie puisque toute tête de message est immédiatement retransmise) soit avec le fait que la configuration de départ est singulière. □

Notation 3 *Pour la simplicité des preuves des lemmes suivants (de 6.2 à 6.4, nous numérotons arbitrairement chaque câble de* 1 *à* L *(le nombre de câbles dans le système) et nous notons* l_j *toute émission sur le câble numéro* j*. Ainsi* l_j *et* l_p *représentent des émissions sur des câbles différents si* $j \neq p$ *et des câbles identiques si* $j = p$*.*

Lemme 6.2 (Unicité) *Entre deux émissions successives de l'unique tête de message sur un même câble, aucun câble ne retransmet deux fois la tête de message.*

Preuve: Considérons une exécution de l'algorithme. Par le lemme 6.1, cette exécution possède une infinité de facteurs commençant et finissant par l'action d'émission l_1 et telle que ces deux actions d'émission sont successives (dans le sens où tous ces facteurs ne contiennent aucune autre action l_1). Intéressons-nous maintenant à la trace restreinte aux actions d'émission d'un tel facteur.

Nous éliminons immédiatement le cas sans intérêt d'une exécution comportant une trace $t = l_1 l_1$ où aucune autre émission que celle de l'unique processeur vers lui-même via l'unique câble n'est possible. En effet, dans ce cas trivial, l'unique tête de message parcourt nécessairement un circuit eulérien.

Notons $t = l_1 l_2 \dots l_k \dots l_n l_k \dots l_1$ la trace que nous considérons. Cette écriture met en évidence que seuls le premier et le dernier élément de la trace t sont identiques, tandis que les n premiers éléments de la trace t sont deux à deux distincts. Plus précisément, l_k est la première répétition d'une émission sur un câble dans la trace.

Afin de prouver le lemme nous allons montrer que l'existence de l_k (c'est à dire d'une répétition) mène à une contradiction avec les hypothèses.

L'émission l_k correspond à l'émission de la tête du message via un câble que nous noterons $c_{i\to j}$. Comme le processeur P_i, au même titre que les autres processeurs, exécute convenablement son algorithme local, s'il a émis deux fois via le câble $c_{i\to j}$ (les deux occurences de l_k) c'est qu'il a reçu $\delta^+(i)+1$ fois la tête du message. Or, s'il a reçu $\delta^+(i)+1$ fois la tête du message, comme le graphe est eulérien, il l'a reçu $\delta^-(i)+1$ fois et par suite il l'a reçu deux fois du même câble. Pour qu'il le reçoive deux fois du même câble, il faut que deux émissions sur ce câble soient survenues. Dans la notation de la trace t les émissions qui ont pu engendrer les retransmissions l_k de la part du processus P_i sont les émissions de l_{k-1} à l_n. Nous venons de montrer que deux d'entre elles sont identiques alors que par hyptohèse elles étaient deux à deux distinctes. Aucune répétition ne peut donc exister et toute trace de longueur $p+1$ commencant et finissant par l_1 s'écrit à permutation près $t = l_1 \dots l_p l_1$. □

Lemme 6.3 (Totalité) *Après que tous les liens ont été visités une fois par la tête de message, entre deux émissions de la tête de message sur un même lien, tous les liens sont visités exactement une fois.*

Preuve: Considérons une exécution de l'algorithme. D'après le lemme 6.1, cette exécution contient une infinité de fois chaque action d'émission *via* un câble. Il est donc possible de décomposer sa trace restreinte aux actions d'émissions sous la forme : $t_0 l_1 t_1 l_1 t_2 \dots l_1 t_n l_1 \dots$ où t_0 est une trace restreinte contenant au moins une fois chacun des $l_{k\in\{1,\dots,L\}}$ et où aucune des $t_{i\geq 1}$ ne contient l_1. D'après le lemme 6.2, il est impossible qu'un des $t_{i\geq 1}$ contiennent deux fois une même action d'emission $l_{k\neq 1}$. Toutes les traces restreintes (sur les actions d'émissions) $t_{i\geq 1}$ sont donc de longueur au plus $L-1$.

Supposons que la trace t_j ($j \geq 1$) est de longueur strictement inférieure à $L-1$ et notons l_p ($p \neq 1$) l'émission qui n'apparait pas dans t_j. Le lemme 6.1 assure que l_p apparaît une infinité de fois dans l'exécution. Il existe donc un plus petit $k > j$ tel que l_p est une émission de la trace t_k. Par ailleurs, comme l_p apparaît par définition dans t_0 il existe aussi un plus grand $m < j$ tel que l_p est une émission de la trace t_m.

En conséquence, l'exécution considérée possède un facteur dont la trace restreinte est $t_m l_1 t_{m+1} \dots t_j \dots t_{k-1} l_1 t_k$ avec l_p ($p \neq 1$) qui n'apparaît dans aucun des $t_{i\in\{m+1,\dots,k-1\}}$ mais

qui apparaît dans t_m et dans t_k :

$$\underbrace{\ldots l_p \ldots}_{t_m} \overbrace{l_1 t_{m+1} \ldots t_j \ldots t_{k-1} l_1}^{\text{aucun } l_p} \underbrace{\ldots l_p \ldots}_{t_k}$$

L'émission l_1 apparaît alors deux fois entre deux émissions successives sur un même câble l_p, ce qui contredit le lemme 6.2.

En conclusion dans la trace restreinte de l'exécution considérée $t_0 l_1 t_1 l_1 t_2 \ldots l_1 t_n l_1 \ldots$, tous les $t_{i\geq 1}$ ne contiennent aucune répétition et sont de taille exactement $L-1$.

Autrement dit, après que tous les liens ont été visités une fois par la tête de message (après t_0), entre deux émissions de la tête de message sur un même lien l_1, tous les liens sont visités exactement une fois (les $t_{i\geq 1}$ contiennent exactement une occurence de chacun des $l_{k\in\{2,\ldots,L\}}$). □

Lemme 6.4 (Légitimité) *Après que tout les liens ont été visités une fois par la tête de message, entre deux émissions de la tête de message sur un même lien, tous les liens sont visités exactement une fois et toujours dans le même ordre.*

Preuve: Considérons une exécution de l'algorithme. D'après le lemme 6.1, cette éxécution contient une infinité de fois chaque action d'émission via un câble. Il est donc possible de décomposer sa trace restreinte aux actions d'émissions sous la forme : $t_0 l_1 t_1 l_1 t_2 \ldots l_1 t_n l_1 \ldots$, où t_0 est une trace restreinte contenant au moins une fois chacun des $l_{k\in\{1,\ldots,L\}}$ et où d'après le lemme 6.3 tous les $t_{i\geq 1}$ contiennent exactement une fois chacun des $l_{k\in\{2,\ldots,L\}}$. Plus précisément, pour $t_j = l_2 l_3 \ldots l_L$ on peut noter t_{j+1} sous la forme $t_{j+1} = l_{\sigma(2)} l_{\sigma(3)} \ldots l_{\sigma(L)}$ où σ est une permutation. Supposons qu'il existe p le plus petit entier de $\{2,\ldots,L\}$ tel que $\sigma(p) \neq p$. Alors il existe un entier q $(p < q \leq L)$ tel que $\sigma(p) = q$.

On peut alors réécrire la trace restreinte d'un des facteurs de l'exécution sous la forme :

$$\underbrace{l_2 l_3 \ldots l_{p-1} l_p l_{p+1} \ldots l_{q-1} l_q \ldots l_L}_{t_j} l_1 \underbrace{l_2 l_3 \ldots l_{p-1} l_q l_{\sigma(p+1)} \ldots l_{\sigma(q-1)} l_p \ldots l_{\sigma(L)}}_{t_{j+1}} l_1$$

Or du point de vue de l_p, le lemme 6.2 n'est pas respecté. En effet entre deux occurences de l_p on trouve deux occurences de l_q. Cette contradiction permet d'affirmer que toute les permutations sont reduites à l'identité et que la trace restreinte de l'exécution s'écrit $t_0(l_1 t_1)^\omega$. Après que tous les liens ont été visités une fois par la tête de message (après t_0), entre deux émissions de la tête de message sur un même lien l_1, tous les liens sont visités exactement une fois et toujours dans le même ordre t_1. □

Théorème 7 *En partant d'une configuration singulière, l'algorithme 5 construit un circuit eulérien et est auto-stabilisant.*

Preuve: Dans le lemme6.4 nous avons prouvé que toute exécution possède une trace restreinte sur les émissions de l'unique messages de la forme $t_0(l_1 l_2 \dots l_L)^\omega$ où t_0 est une trace finie. En conséquence un circuit eulérien parcourant les liens numéro 1 à L est construit infiniment souvent après un nombre de retransmissions fini du message. □

Définition 6.7 (Configuration Eulérienne) *Une configuration c est* eulérienne *si et seulement si elle est singulière et si pour toute exécution $e \in \mathcal{E}_c$, l'unique tête de message m traverse un circuit eulérien au moins une fois. Un processeur P_i est* eulérien *si et seulement si son état apparaît dans une configuration eulérienne.*

6.4 Applications

Dans cette section, nous proposons deux applications de l'algorithme de construction de circuit eulérien d'une part et des techniques de propagation de marqueur mentionnées. La première application consiste à distinguer un processeur sur le réseau (tâche d'élection d'un chef) et la deuxième consiste à construire un mécanisme permettant à chaque processeur de déterminer l'ensemble des processeurs pésents (tâche de recensement).

Anneau Virtuel Chacune de ces applications fait l'hypothèse d'une exécution sur un réseau en anneau unidirectionnel. Dans la pratique, nous considérons que soit le graphe de communication est effectivement un anneau unidirectionnel, soit le graphe de communication est un graphe eulérien dans lequel on exécute simultanément le protocole de construction de circuit eulérien (algorithme 5). Nous supposons en outre que l'algorithme de niveau supérieur (élection ou recensement) n'est exécuté par l'algorithme de niveau inférieur (l'algorithme de circuit eulérien) que si la variable $Chemin_i$ vaut 0. Ceci signifie que, dans le cas où le réseau n'est pas un anneau unidirectionnel, alors l'unique message est transmis à l'algorithme de haut niveau sur chaque processeur une unique fois lors de chaque circuit eulérien. De cette manière, l'algorithme 5 construit un circuit virtuel où chaque processeur est visité *exactement* une fois lors de chaque traversée. Dans la suite, nous considérons sans perte de généralité, que le réseau est un anneau unidirectionnel, de telle sorte qu'un *tour* dénote un circuit eulérien.

Complétude des Messages De plus, les configurations de départ que nous considérons sont les configurations où tous les messages sont bien formés, c'est à dire qu'ils comprennent un nombre adéquat de bits de manière à former un ou plusieurs enregistrements au sens défini par le programme. Ces configurations sont appelées dans la suite des *configurations complètes.*

6.4.1 Election d'un Chef

La tâche d'élection est l'un des problèmes répartis les plus fondamentaux, et de nombreuses solutions auto-stabilisantes ont été présentées, parmi celles-ci on peut citer [AKY90], [AKM+93], [CYH91] et [DIM93]. La plupart de ces solutions sont basées sur la construction

auto-stabilisante d'un arbres couvrant, dont la racine devient le processeur élu. Notre approche est basée sur les identifiants des processeurs et ne requiert pas de construction d'arbre.

Algorithme

Chacun des processeurs P_i de l'anneau unidirectionnel est doté d'un identifiant unique Id_i (et donc un ordre total peut être induit sur ces identifiants) et d'une variable booléenne déterminant s'il est l'élu. Le critère d'élection de l'algorithme est que le processeur dont l'identifiant est le plus élevé est élu et que les autres processeurs ne sont pas élus.

L'algorithme utilise des messages de structure $\langle\ Id\ ,\ Nb\ \rangle$, où :

- *Id* est un entier positif suffisamment grand pour contenir n'importe quel identifiant de processeur. Ce champ est utilisé pour résoudre l'élection en choisissant l'identifiant le plus grand du réseau. *Id* est stocké avec les bits de poids fort en premier.
- *Nb* est un entier positif de taille égale à celle du champ *Id*. Il dénote le nombre de processeurs qui ont déjà reçu le message. Notons que les processeurs ne connaissent pas la taille du réseau. Le champ *Nd* est stocké avec ses bits de poids faible en premier.

Remarque 6.1 *Le champ* Id *est stocké avec les bits de poids fort en premier, donc un algorithme cut-though peut effectuer des comparaisons et des remplacement à la volée avec un tampon de taille constante (*1*) et aucun marqueur.*

Remarque 6.2 *Le champ* Nb *est stocké avec les bits de poids faible en premier, donc un algorithme à délai borné peut effectuer des opérations d'incrément à la volée avec un tampon de taille constante (*1*) et aucun marqueur.*

En outre, chaque processeur P_i possède une unique variable booléenne $Leader_i$, qui vaut *vrai* si le processeur est élu et *faux* sinon. Le but de l'algorithme est de garantir qu'au bout d'un temps fini, un unique processeur P_i ait sa variable $Leader_i$ à vrai dans tout le réseau.

Algorithme 6 (Election) *Algorithme du processeur P_i sur réception du message* $\langle \mathrm{Id}_m, \mathrm{Nb}_m \rangle$

Si $\mathrm{Id}_m \leq \mathrm{Id}_i$ *Alors*
  $\mathrm{Leader}_i \leftarrow$ vrai
  Retransmettre Id_i *{comme* Id_m*}*
  Retransmettre 0 {comme Nb_m*}*
Sinon {$\mathrm{Id}_m > \mathrm{Id}_i$ *}*
  $\mathrm{Leader}_i \leftarrow$ faux
  Retransmettre Id_m *{comme* Id_m*}*
  Retransmettre $\mathrm{Nb}_m + 1$ *{comme* Nb_m*}*
  Si $\mathrm{Nb}_m > \mathrm{Id}_m$ *Alors*
    Réinitialiser le message {avec un marqueur Z}
  FinSi
FinSi

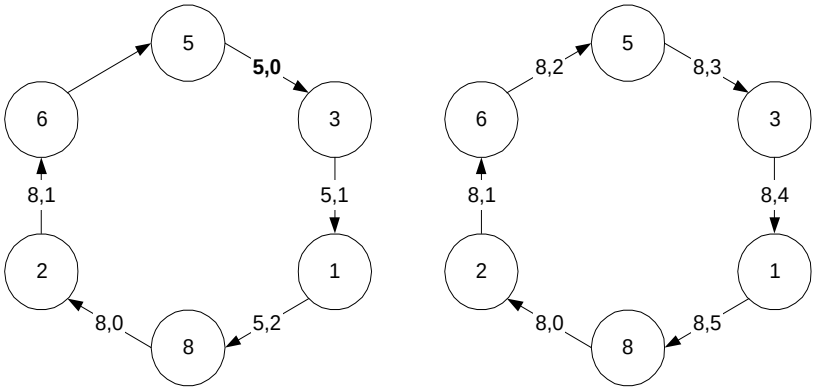

FIG. 6.10 – Election d'un Chef à Partir d'un Nouveau Message.

Contraintes de Délai Borné Notons tout d'abord que cet algorithme satisfait les contraintes du routage à délai borné, même avec un *tampon* de taille constante (supérieure à 3) sur chaque processeur.

Pour chaque message le processeur P_i envoie un nouveau message sur un câble de sortie. Le champ Id_m d'un message étant présenté avec les bits de poids fort en premier, une comparaison avec Id_i peut être effectuée à la volée avec un *tampon* de taille 1. Le champ Nb_m d'un enregistrement étant présenté avec les bits de poids faible en premier, il peut être incrémenté à la volée (si $Id_m > Id_i$) ou rémis à zéro (si $Id_m \leq Id_i$) avec un *tampon* de taille 1.

Au final, les opérations de comparaison et d'incrément nécéssitent un *tampon* de taille 1, la réinitialisation au moyen d'un marqueur Z nécessite d'accroître la taille du *tampon* de 1 et le retardement probabiliste nécessite d'accroître la taille du *tampon* de 1 une nouvelle fois. Au total, nous avons besoin d'un *tampon* capable de stocker 3 éléments de signal.

Exemple d'Exécution La figure 6.10 montre une exécution du système après que le processeur 5 a envoyé un nouveau message (donc un message $\langle 5, 0\rangle$). Le champ Nb_m est successivement incrémenté par les processeurs situés sur le chemin du message dont l'identifiant est inférieur au champ Id_m (les processeurs 3 et 1). Par la suite le processeur 8 constate que son identifiant est supérieur à 5 et il initie à son tour un nouveau message $\langle 8, 0\rangle$. Comme 8 est l'identifiant maximum sur le réseau, seul le champ Nb_m est maintenant modifié par les différents processeurs. Les processeurs 2, 6, 5, 3 et 1 incrémentent ce champ, alors que le processeur 8 le réinitialise.

La figure 6.11 présente une exécution du système dans le cas où l'identifiant contenu dans le message en transit (soit 9) n'est pas un identifiant d'un processeur du réseau. Le champ Nb_m du message est alors incrémenté par tous les processeurs du réseau, jusqu'à devenir supérieur ou égal au champ Id_m. Cette incohérence est détectée par le processeur 1 dans l'exemple, qui émet un marqueur Z afin de réinitialiser le message. Comme la capacité des

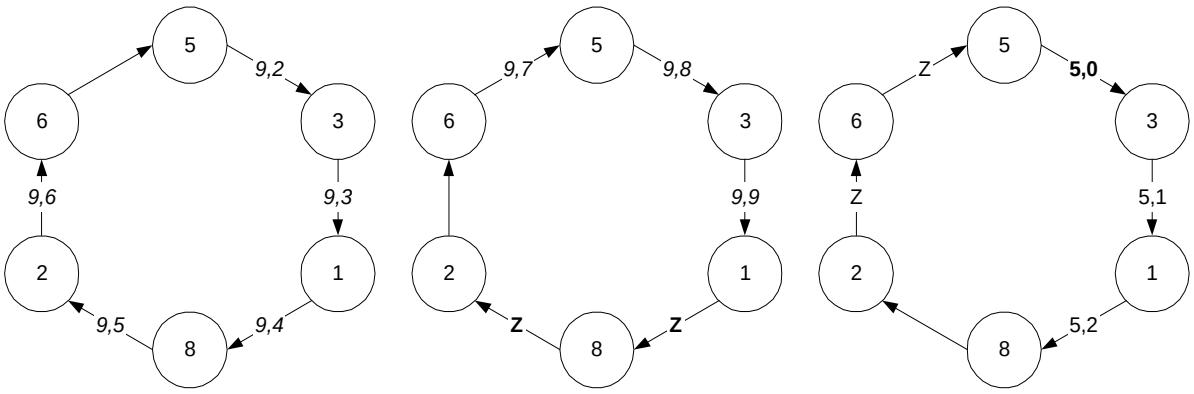

FIG. 6.11 – Suppression d'un Chef Inexistant.

Buffers des processeurs est fixée, ce marqueur peut mettre un certain temps avant de parvenir en début de message. Quand la réinitialisation est totalement effectuée, un nouveau message peut être émis par un processeur, et le même mécanisme que dans l'exemple précédent peut être utilisé.

Preuve de Stabilisation

Lemme 6.5 *En partant d'une configuration complète, le champ* Id_m *d'un message est fatalement égal à un identifiant de processeur existant.*

Preuve: Supposons que ce ne soit pas le cas : le champ Id_m d'un message est différent de tous les identifiants de tous les processeurs du réseau et n'est jamais changé (c'est à dire que les lignes 1 à 4 de l'algorithme 6 ne sont jamais exécutées). Ceci implique que cet identifiant est supérieur à tout les identifiants du réseau, sinon l'un des processeurs l'aurait remplacé par son propre identifiant (voir la ligne 5 de l'algorithme 6). Alors chaque processeur incrémente le champ Nb_m du message, qui devient fatalement plus grand que le champ Id_m, et alors un marqueur Z est retransmis. Fatalement, ce marqueur arrive en tête du message et un nouveau message de type $\langle Id_k, 0\rangle$ est envoyé par un processeur P_k. Puisque les processeurs retransmettent le champ Id_m d'un message ou l'échangent par le leur (voir les lignes 3 et 7 de l'algorithme 6), aucun identifiant n'appartenant à aucun processeur du réseau ne peut plus apparaître dans le champ Id_m. □

Le lemme suivant complète la preuve de convergence en accord avec la définition des configurations légitimes.

Définition 6.8 (Configuration Légitime) *Une configuration est* légitime *pour l'algorithme 6 si et seulement si :*

1. *le processeur avec le plus grand identifiant a sa variable* Leader *à* vrai *et tous les autres processeurs ont leur variable à* faux *;*

2. *la configuration est complète ;*
3. *pour tout message m, le champ* Id_m *est égal à l'identifiant de processeur le plus grand du réseau, et le champ* Nb_m *est plus petit que le champ* Id_m*.*

Définition 6.9 (Message Légitime) *Un message est* légitime *si et seulement si son champ* Id_m *est égal à l'identifiant de processeur le plus grand du réseau et son champ* Nb_m *est inférieur à son champ* Id_m*.*

Lemme 6.6 *En partant d'une configuration complète c, chaque nouveau message (c'est à dire un message de la forme* $\langle \mathrm{Id}, 0 \rangle$*, où* Id *est un identifiant de processeur valide) devient légitime dans toute exécution $e \in \mathcal{E}_c$.*

Preuve: En un tour, un nouveau message visite chaque processeur exactement une fois, et donc le champ Id_m devient égal à l'identifiant maximal de processeur Max du réseau. En un tour, le champ Nb_m du message peut être incrémenté au plus de $Max-1$, et le champ Nb_m est remis à 0 par le processeur dont l'identifiant est Max. Dans tous les cas, le champ Id_m du message vaut Max, et le champ Nb_m reste inférieur à Max, donc le message est et reste légitime. □

Lemme 6.7 *En partant d'une configuration complète c, chaque message devient légitime dans toute exécution $e \in \mathcal{E}_c$.*

Preuve: Par le lemme 6.5, le champ Id_m de tout message m est fatalement égal à un identifiant de processeur. Soit $c' \in \mathcal{C}$ une configuration où cela est vrai. En un tour à partir de c', si le message est réinitialisé, alors par le lemme 6.6 il devient légitime et le reste. Si il n'est pas remis à zéro, cela signifie qu'il a visité tous les processeurs et que chacun d'entre eux a éventuellement procédé au remplacement du champ Id_m du message par son propre identifiant, et que ce champ vaut actuellement Max, l'identifiant maximal du réseau. A chaque fois qu'un processeur remplace le champ Id_m du message par son propre identifiant, il réinitialise le message, et par le lemme 6.6, le message devient nécessairement légitime. □

Lemme 6.8 (Election d'un Chef) *En partant d'un configuration complète où chaque message m est légitime, une configuration légitime est nécessairement atteinte.*

Preuve: En partant d'une configuration complète où chaque message est légitime, en un tour tous les processeurs ont pris connaissance de l'identifiant maximal du réseau et mis à jour leur variable $Leader_i$ en conséquence. Par suite, une configuration légitime est atteinte. □

A partir des lemmes précédents, il est possible de garantir l'auto-stabilisation du système.

Théorème 8 (Election) *L'algorithme 6 effectue une élection d'un chef sous un routage à délai borné de manière auto-stabilisante.*

Complexité

Lemme 6.9 (Complexité en Espace des Messages) *Un message légitime de l'algorithme 6 occupe un espace de $MS_{EL} = O(\log_2 N)$ bits.*

Preuve: Chaque message légitime possède 2 champs. Chaque champ varie de 0 à $N-1$ (soit $\log_2 N$ bits). Au total :

$$\begin{aligned} MS_{EL} &= 2 \times \log_2 N \\ &= O(\log_2 N) \end{aligned}$$

□

Lemme 6.10 (Complexité en Espace des Processeurs) *Chaque processeur exécutant l'algorithme 6 possède une mémoire de $NS_{EL} = O(1)$ bits.*

Preuve: Chaque processeur possède un *tampon* de 3 bits, ainsi qu'une variable booléenne (soit 1 bits). Donc nous avons

$$\begin{aligned} NS_{EL} &= 4 \\ &= O(1) \end{aligned}$$

□

Dans le cas des réseaux en anneau, le temps nécessaire à la complétion d'un tour, ou *temps de tour* est donné par la relation $T_r = \sum_{i=0}^{N-1} \left(T_{d_i} + T_{p_{i \leftarrow i+1 \bmod N}}\right)$. Dans la suite, nous supposons que le temps de propagation est le même pour tous les liens et est égal à T_1.

Lemme 6.11 (Temps de Rotation) *Le temps de rotation de l'algorithme 6 est en $O(N)$.*

Preuve:

$$\begin{aligned} T_{r_{EL}} &= \sum_{i=0}^{N-1} \left(T_{d_i} + T_{p_{i \leftarrow i+1 \bmod N}}\right) \\ &\approx \sum_{i=0}^{N-1} \left(\frac{3}{C} + T_1\right) \\ &\approx N\left(\frac{3}{C} + T_1\right) \\ &\approx \frac{3 \times N}{C} + N \\ &= O(N) \end{aligned}$$

□

Lemme 6.12 (Temps de Stabilisation) *Le temps de stabilisation de l'algorithme 6 est en $O(N)$.*

Preuve: S'il n'y a aucun message initialement dans le réseau, alors nous pouvons supposer qu'un temporisateur est activé après un temps $T_{r_{EL}}$. Supposons qu'initialement, il y ait un message incorrect dans le réseau. Au moins un processeur va réinitialiser le message correct en un temps au plus $T_{r_{EL}}$. Mais dans notre algorithme, l'utilisation des marqueurs fait qu'il est possible qu'un délai supplémentaire de $\frac{\log_2 N}{3}$ soit nécessaire pour amener le marqueur au début du message. Après que ce message a été détruit et un nouveau message initié, il faut

un tour supplémentaire soit un temps $T_{r_{EL}}$ avant que le message m devienne légitime. Dans le pire des cas, le temps est

$$\begin{aligned} T_{Stab_{EL}} &= 2T_{r_{CT}} + \frac{\log_2 N}{3} \\ &= O(N) \end{aligned}$$

□

6.4.2 Recensement

Le problème du *recensement* est dérivé de celui de mise à jour de topologie en supprimant le prérequis d'information de localisation. Le problème de mise à jour de topologie a été résolu de manière auto-stabilisante à de multiples reprises [DT98a, Dol97, DH97, Mas95, SG89]. Cependant, tous ces protocoles ne sont valides que dans le modèle à passage de messages. Un algorithme auto-stabilisant de recensement a été présenté dans [DT98a], mais l'espace requis est relativement élevé, de l'ordre de la taille du réseau.

Spécification du Problème

Chaque processeur $P_v \in P$ du système $\mathcal{S} = (P, L)$ possède un identifiant Id_v. Un utilisateur externe au système peut exécuter sur chaque processeur du réseau la fonction Existe. Le résultat de cette fonction Existe(v) est *vrai* si et seulement si $v \in P$. Cette fonction possède la signature suivante :

$$\begin{array}{rcccl} \mathsf{Existe} & : & P & \rightarrow & \{vrai, faux\} \\ & & v & \mapsto & \mathsf{Existe}(v) \end{array}$$

Algorithme

Dans cette section, nous proposons un algorithme auto-stabilisant (appelé ci-après l'algorithme $\mathcal{CSF}$) pour résoudre le problème du recensement dans un réseau en anneau. Nous donnons une solution qui fonctionne dans le modèle de routage à délai borné.

Le principe de base est le suivant : au bout d'un temps fini, il y a au moins un message dans l'anneau qui collecte les identifiants des processeurs et qui enlève les identifiants qui ne correspondent pas à des processeurs existants. Chaque proceseur recevant un tel message effectue des corrections mineures quand il détecte des inconsistances.

Chaque message m contient 0 ou plusieurs enregistrements. Chacun de ces enregistrements contient deux champs : Id_m, un identifiant de processeur, et Nb_m, le nombre de processeurs visités par ce champ depuis qu'il a visité le processeur d'identifiant Id. Le champ Nb est utilisé pour assurer que tous les enregistrements correspondent à des processeurs existants et *seulement* à ceux-ci. L'enregistrement correspondant au processeur P_i dans le message m est dénoté par $m(i) = \langle Id_m^i, Nb_m^i \rangle$.

Notre algorithme effectue trois tâches essentielles :

1. *Ajout des informations manquantes.* Si un processeur P_i détecte que les informations le concernant n'existe pas dans le message, il les ajoute en fin de message.
2. *Correction des champs* Nb^i_m. La valeur de chaque champ Nb^i_m doit satisfaire les deux conditions suivantes : *(i)* elle est égale au maximum des Nb^i_m et *(ii)* elle est égale à la taille du message (en nombre d'enregistrements) moins 1. Si l'une de ces deux conditions est fausse, cela signifie que le message contient un ou plusieurs enregistrements erronés, et le processeur P_i envoie un marqueur Z afin de réinitialiser le message.
3. *Implantation de la fonction* Existe. Cet aspect n'est pas décrit en détail ici, mais l'implantation d'une telle fonction est facile. Il est possible soit de garder une copie locale du message sur chaque processeur à chaque nouvelle arrivée d'un message (mais le coût en mémoire est alors proportionnel à la taille du réseau), soit d'attendre qu'un message arrive sur le processeur où la fonction Existe a été appelée puis de vérifier que l'identifiant passé en paramètre de la fonction est bien présent dans le message en transit (ceci peut être effectué même avec les contraintes du délai borné). C'est cette deuxième option que nous choisissons pour calculer la complexité en mémoire de notre algorithme.

Algorithme 7 (Recensement) *Algorithme du processeur P_i sur réception de message*

Local JeSuisLa=faux
Local JeSuisLaDeuxFois=faux
Local Taille = 0
Local NbMax = 0
PourChaque $\langle \mathrm{Id}_m, \mathrm{Nb}_m \rangle$ *de* $\langle \mathrm{Id}_1, \mathrm{Nb}_2 \rangle \cdots \langle \mathrm{Id}_k, \mathrm{Nb}_k \rangle$ *{le message}*
    Taille← Taille + 1 *{calcul de la taille du message}*
    Si $\mathrm{Id}_m = \mathrm{Id}_i$ *Alors*
        Si JeSuisLa = vrai *Alors*
            JeSuisLaDeuxFois ← vrai
        Sinon
            JeSuisLa ←vrai
        FinSi
        Retransmettre Id_i *{comme* Id_m*}*
        Retransmettre 0 {comme Nb_m*}*
    Sinon $\{\mathrm{Id}_m \neq \mathrm{Id}_i\}$
        Retransmettre Id_m *{comme* Id_m*}*
        Retransmettre $\mathrm{Nb}_m + 1$ *{comme* Nb_m*}*
        Si $\mathrm{Nb}_m + 1 >$ NbMax *Alors*
            NbMax ← $\mathrm{Nb}_m + 1$
        FinSi
    FinSi
FinPourChaque
Si (NbMax > Taille) *Ou* (JeSuisLaDeuxFois = vrai) *Alors*

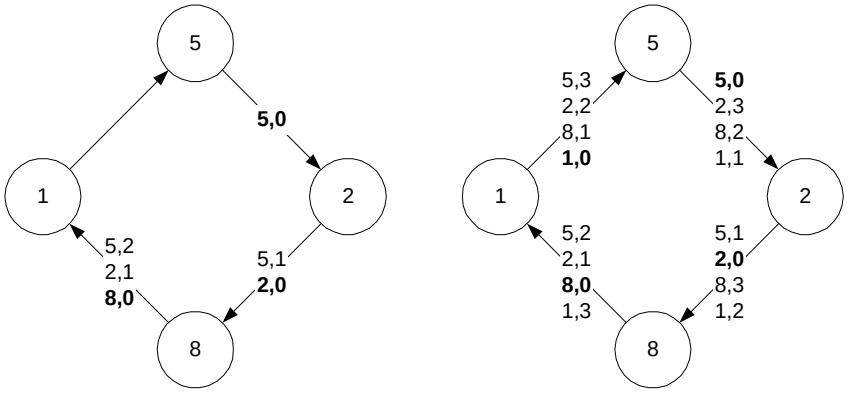

FIG. 6.12 – Recensement à Partir d'un Nouveau Message.

Réinitialiser le message {avec un marqueur Z}
Sinon
Si JeSuisLa = faux *Alors*
Ajouter Id_i *{comme* Id_m*}*
Ajouter 0 *{comme* Nb_m*}*
FinSi
FinSi

Les Contraintes du Délai Borné Notons tout d'abord que cet algorithme satisfait les contraintes du routage à délai borné, même avec un *tampon* de taille constante (supérieure à 3) sur chaque processeur.

Pour chaque nouvel enregistrement le processeur P_i envoie un nouvel enregistrement sur un câble de sortie. La comparaison entre le champ Id_m d'un enregistrement et Id_i ayant uniquement trait à une égalité/différence, elle peut être effectuée à la volée avec un *tampon* de taille 1. Le champ Nb_m d'un enregistrement étant présenté avec les bits de poids faible en premier, il peut être incrémenté à la volée (si $Id_m \neq Id_i$) ou rémis à zéro (si $Id_m = Id_i$) avec un *tampon* de taille 1. L'ajout éventuel d'enregistrements ou la réinitialisation du message sont effectués une fois qu'il n'existe plus aucun enregistrement dans le message (un élément de signal `N` a été lu sur le câble entrant), donc les nouveaux symboles sont ajoutés à la fin du message.

Au final, les opérations de comparaison et d'incrément nécessitent un *tampon* de taille 1, la réinitialisation au moyen d'un marqueur `Z` nécessite d'accroître la taille du *tampon* de 1 et le retardement probabiliste nécessite d'accroître la taille du *tampon* de 1 une nouvelle fois. Au total, nous avons besoin d'un *tampon* capable de stocker 3 éléments de signal.

Exemple d'exécution La figure 6.12 présente l'exécution de notre algorithme à partir d'une configuration où un unique nouveau message est envoyé par le processeur 5 (il contient

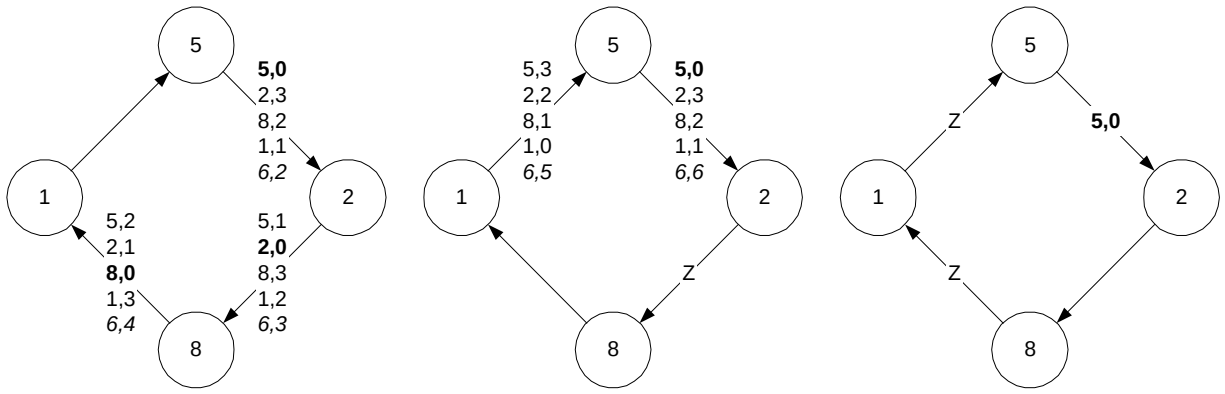

FIG. 6.13 – Elimination d'un Identifiant Inexistant.

donc un message $\langle 5, 0\rangle$). Dès lors, chacun des processeurs sur le chemin du message (les processeurs 2, puis 8 et enfin 1) ajoute en fin de message leur propre identifiant, en ayant soin d'incrémenter la distance (le champ Nb_m) de chacun des enregistrements dont il ne possède pas le même identifiant. Au terme d'un tour d'anneau, le message contient tous les identifiants du système et peut être utilisé pour implanter la fonction Existe.

La figure 6.13 présente une exécution du système suite à une défaillance transitoire qui a placé un identifiant de processeur et sa distance (l'enregistrement $\langle 6, 2\rangle$) dans l'unique message du système, alors que le processeur correspondant n'existe pas dans le système. Chacun des processeurs sur le chemin de ce message incrémente le champ Nb_6 jusqu'à ce que celui-ci dépasse la taille du message (il vaut 6 alors que la taille du message est de 5 enregistrements, donc il est visiblement erroné). Un marqueur Z est alors envoyé pour réinitialiser le message. Comme les *Buffers* des processeurs sont de taille fixée, le marqueur peut prendre un certain temps avant d'arriver au début du message. Au final, le processeur 5 reçoit un message commençant par l'élément de signal Z, et il envoie un nouveau message, comme lors de l'exemple précédent.

Preuve de Correction

Nous commençons par définir les configurations légitimes pour l'algorithme 7.

Définition 6.10 (Message Légitime) *Un enregistrement* $m(i) = \langle \mathrm{Id}^i_m, \mathrm{Nb}^i_m \rangle$ *d'un message* m *est* légitime *si ses champs satisfont les propriétés suivantes :* (i) *le champ* Id^i_m *est égal à* Id_j *pour un processeur* P_j *du réseau ;* (ii) *le champ* Nb^i_m *est égal au nombre de processeurs visités par* m *depuis le processeur* P_j*, c'est à dire la distance à* P_j*. Un message* m *est* légitime *si et seulement si il possède exactement* N *enregistrement légitimes dont les* Ids *sont différents.*

Notons qu'un message légitime contient un enregistrement légitime pour chacun des processeurs du réseau. Donc il contient N valeurs distinctes de *Id*.

Définition 6.11 (Configuration Légitime) *L'ensemble des configurations légitimes $\mathcal{L}_S$ de $\mathcal{S}$ de l'algorithme 7 est tel que dans chaque configuration $c \in \mathcal{L}_S$, les conditions suivantes sont vérifiées :* (i) *il y a exactement un message dans le réseau ;* (ii) *le message est légitime.*

Nous devons tout d'abord nous assurer que de telles configurations sont possibles avec nos hypothèses, de manière à ce qu'il existe une solution à la spécification que nous avons donné.

Lemme 6.13 *Le réseau peut contenir un message légitime.*

Preuve: Un message légitime contient N enregistrements. Donc la taille d'un message légitime est $ms = N \times$`RecSize`.Chaque processeur possède un *tampon* dont la taille est supérieure à celle d'un enregistrement. Chaque lien de communication peut contenir au moins 1 élément de signal. Donc le réseau peut contenir $ns = N \times$ (`RecSize`$+2$) éléments de signal. Pour $N \geq 2$, $ns > ms$. □

Lemme 6.14 (Clôture) *L'ensemble $\mathcal{L}_\mathcal{S}$ des configurations légitimes est clos.*

Preuve: Prouvons tout d'abord que tout message légitime reste légitime. Soit m un message légitime arrivant sur le processeur P_k. Puisque m est légitime, chaque enregistrement Nb_m^i est égal à la distance au processeur P_i. Donc le champ Nb_m^k est égal à $N-1$ et est le maximum parmi les différents champs Nb_m^i. Le processeur P_k met Nb_m^k à 0 et incrémente les autres champs $Nb_m^{i \neq k}$ de 1. Donc le message reste légitime.

Prouvons maintenant qu'en partant d'une configuration légitime, aucun message ne peut être perdu. Dans une configuration légitime, tous les messages contiennent N enregistrements différents (un par processeur). Donc il n'y a pas de marqueurs `Z` dans les messages, et aucun processeur ne peut adjoindre son identifiant dans l'un de ces messages. □

Il reste à prouver qu'en partant d'une configuration arbitraire du système, toute exécution de l'algorithme 7 atteint une configuration de $\mathcal{L}_S$.

Définissons $\mathcal{C}_1$ comme l'ensemble des configurations telles qu'il existe exactement un message dans le réseau, et que chaque message contienne seulement des identifiants qui correspondent à des identifiants des processeurs du système. De plus, tout Id_m^i apparait dans un message une unique fois. Il est clair que $\mathcal{L}_S \subset \mathcal{C}_1 \subset \mathcal{C}$. Donc, pour prouver la propriété de convergence de l'algorithme 7, nous devons prouver que $\mathcal{L}_S \triangleleft \mathcal{C}_1 \triangleleft \mathcal{C}$.

Lemme 6.15 $\mathcal{C}_1 \triangleleft \mathcal{C}$

Preuve: Assurons nous tout d'abord que chaque message qui n'est pas réinitialisé satisfait au bout d'un temps fini la spécification de $\mathcal{C}_1$:

1. m contient des Ids qui ne correspondent pas à un identifiant de processeur. Soit Id_m^k un tel identifiant, le champ Nb_m^k est alors incrémenté par chaque processeur du système un fois par tour. Comme après un tour, plus aucun processeur n'ajoute de nouvel enregistrement au message, le champ Nb_m^k finit par exéder la taille (en nombre d'enregistrements) du message, et le message m est réinitialisé au moyen d'un marqueur `Z`. Le cas 2 est alors vérifié.

2. m contient des `Id`s qui sont tous valides. Les identifiants dupliqués sont supprimés par retransmission d'un marqueur `Z`. Ceci a pour effet d'initier un nouveau message. Alors fatalement aucun message ne contient plus d'identifiant incorrect ou dupliqué. Si au moins un message n'est pas détruit, alors nous atteignons une configuration $c \in \mathcal{C}_1$.

Prouvons maintenant qu'en partant d'une configuration $c \in \mathcal{C}_1$, le système n'atteint jamais une configuration où il n'y a aucun message dans le réseau. Supposons le contraire : il existe une exécution e, où il existe un message m qui est supprimé au cours de l'exécution. Ce message ne peut être supprimé que dans l'un des deux cas suivants :

1. Le processeur P_i est en train de retransmettre m. D'après le lemme 6.13, il y a suffisamment d'espace pour contenir un message légitime. Donc cette situation ne peut survenir puisque tous les messages possibles de $\mathcal{C}_1$ ne sont pas plus grands qu'un message légitime.
2. Le processeur P_i est en train d'ajouter un enregistrement à m. Cette situation ne peut survenir seulement dans le cas où un autre enregistrement (celui de P_i) ne peut être ajouté à m. D'après le lemme 6.13, il y a suffisamment d'espace pour insérer un nouvel enregistrement pour P_i.

□

Lemme 6.16 $\mathcal{L}_S \triangleleft \mathcal{C}_1$.

Preuve: Tout d'abord, nous prouvons qu'un message possédant des enregistrement non-légitimes ne peut subsister à jamais. Supposons q'un message m ne possède pas d'identifiants pour des processeurs non-existants. Mais alors $Nb_m^i \neq Taille - 1$. Ce message va fatalement atteindre le processeur P_i et ce dernier va mettre Nb_m^i à 0. Par suite, l'enregistrement du processeur P_i dans le message m devient légitime. Tous les autres enregistrements de m deviennent fatalement légitimes de manière similaire.

Prouvons maintenant que tout message non-légitime ne peut subsister à jamais. Soit m un message avec $n < N$ enregistrements légitimes différents et pas d'enregistrement pour le processeur P_i. Ce message atteint fatalement P_i. Par la suite, P_i ajoute son propre enregistrement légitime en incrémentant n de 1. Si n valait $N-1$ avant que l'enregistrement de P_i soit adjoint, alors m devient un message légitime. Sinon, m atteint un nouveau processeur et le même mécanisme se reproduit jusqu'à ce que tous les enregistrements de tous les processeurs du réseau soit ajoutés à m. □

Théorème 9 (Recensement) *L'algorithme 7 effectue un recensement de manière autostabilisante sous un routage à délai borné.*

Preuve: D'après les lemmes 6.14, 6.15 et 6.16. □

Complexité

La complexité en espace sur chaque processeur est améliorée d'un facteur N dans le modèle à délai borné par rapport au modèle à passage de messages. Ce résultat est particulièrement intéressant pour des réseaux de grande taille, où une approche classique (c'est à dire un passage de messages) résulterait en une consomation excessive de mémoire.

Lemme 6.17 (Complexité en Espace des Messages) *Un message légitime de l'algorithme 7 occupe un espace de* $MS_{RE} = O(N\log_2 N)$ *bits.*

Preuve: Chaque message légitime possède N enregistrements. Chaque enregistrement est composé d'un champ Id_m^i, qui varie de 0 à $N-1$ (soit $\log_2 N$ bits), et d'un champ Nb_m^i, qui varie de 0 à $N-1$ (soit $\log_2 N$ bits). Par suite,

$$\begin{aligned} MS_{RE} &= N(\log_2 N + \log_2 N) \\ &\approx 2N \log_2 N \\ &= O(N\log_2 N) \end{aligned}$$

□

Lemme 6.18 (Complexité en Espace des Processeurs) *Chaque processeur exécutant l'algorithme 7 possède une mémoire de* $NS_{RE} = O(\log_2 N)$ *bits.*

Preuve: Chaque processeur possède un *tampon* de 3 bits, deux variables booléennes (soit 2 bits) et deux variables pouvant varier de 0 à $N-1$ (soit $2 \times \log_2 N$ bits). Donc nous avons

$$\begin{aligned} NS_{RE} &\approx 2\log_2 N \\ &= O(\log_2 N) \end{aligned}$$

□

Lemme 6.19 (Temps de Rotation) *Le temps de rotation de l'algorithme 7 est en* $O(N)$.

Preuve:

$$\begin{aligned} T_{r_{RE}} &= \sum_{i=0}^{N-1} \left(T_{d_i} + T_{p_{i\leftarrow i+1 \bmod N}}\right) \\ &\approx \sum_{i=0}^{N-1} \left(\frac{3}{C} + T_1\right) \\ &\approx N\left(\frac{3}{C} + T_1\right) \\ &\approx \frac{3\times N}{C} + N \\ &= O(N) \end{aligned}$$

□

Lemme 6.20 (Temps de Stabilisation) *Le temps de stabilisation de l'algorithme 7 est en* $O(N \log_2 N)$.

Preuve: S'il n'y a aucun message initialement dans le réseau, alors nous pouvons supposer qu'un temporisateur est activé après un temps $T_{r_{RE}}$. Supposons qu'initialement, il y ait un message incorrect dans le réseau. Au moins un processeur va réinitialiser le message correct en un temps au plus $T_{r_{RE}}$. Mais dans notre algorithme, l'utilisation des marqueurs fait qu'il est possible qu'un délai supplémentaire de $\frac{N \log_2 N}{3}$ soit nécessaire pour amener le marqueur au début du message. Après que ce message a été détruit et un nouveau message initié, il faut un tour supplémentaire soit un temps $T_{r_{RE}}$ avant que le message devienne légitime. Dans le pire des cas, le temps est

$$\begin{aligned} T_{Stab_{RE}} &= 2T_{r_{RE}} + \frac{N \log_2 N}{3} \\ &= O(N \log_2 N) \end{aligned}$$

□

6.5 Résumé

Dans ce chapitre, nous avons placé au même niveau le protocole de communication utilisé par les processeurs pour communiquer entre eux et l'algorithme implantant une solution à un problème particulier. Ceci a rendu possible un gain substantiel tant au niveau de la mémoire utilisée par chacun des processeurs que du délai intervenant dans une communication d'informations passant par des processeurs internédiaires.

Nous avons proposé plusieurs outils facilitant l'écriture des algorithmes auto-stabilisants dans ce contexte. En particulier, une technique consistant à utiliser des signaux de valence supérieure à 2 rend possible la réinitialisation d'un message (par exemple suite à la détection d'une défaillance transitoire) malgré que celui-ci ait déjà été retransmis. Par ailleurs, nous avons présenté un algorithme auto-stabilisant de construction de circuit eulérien qui rend naturelle l'extension aux réseaux eulériens des algorithmes prévus pour des anneaux unidirectionnels. En application, nous avons présenté deux algorithmes auto-stabilisants, l'un résolvant la tâche d'élection d'un chef, l'autre la tâche du recensement.

Conclusion

Quand un système réparti est sujet à des défaillances transitoires qui modifient arbitrairement les états des processeurs et le contenu des liens qui le constituent, il est crucial de pouvoir retrouver un comportement correct au bout d'un temps fini. L'*auto-stabilisation* est une technique offrant une telle garantie, mais en général au prix de ressources importantes.

Dans cette thèse, notre démarche a consisté à minimiser ces ressources lorsque cela était possible. Nous avons distingué deux contextes représentatifs pour calculer le coût d'un algorithme auto-stabilisant : le système en l'absence de défaillances et le système en présence de défaillances. En l'absence de défaillances, le *surcoût* de l'algorithme auto-stabilisant rend compte de la mémoire et du temps de calcul supplémentaires attribuées spécifiquement à la détection puis correction d'erreurs. En présence de défaillances, le *temps de stabilisation* est le principal critère d'efficacité, puisqu'un système non stabilisant peut très bien ne jamais retrouver un comportement correct.

Nous avons tout d'abord présenté informellement puis formellement les systèmes répartis et l'auto-stabilisation, en faisant systématiquement apparaître que les hypothèses courantes dans le domaine pouvaient être hiérarchisées. Cette hiérarchie rend compte de l'idée intuitive que si un système supporte des hypothèses laxistes (atomicité fine, exécution non équitables, topologie quelconque), alors il supporte de fait des hypothèses plus restrictives (atomicité composée, exécution équitables, topologie eulérienne).

Dans ce modèle, nous avons développé le concept de détecteur de défaillances transitoires, des oracles appelés par les processeurs du système, qui indiquent si des défaillances transitoires sont survenues, en un temps constant. L'implantation que nous avons proposée pour ces détecteurs nous a permis de classifier les problèmes classiques des systèmes répartis suivant les ressources spécifiques nécessaires à la détection d'une défaillance transitoire en faisant abstraction de l'algorithme résolvant le problème.

Dans le cas des tâches statiques, ceci nous a donné plusieurs bornes supérieures relatives à la mémoire à utiliser lors de la détection d'erreurs. Dans le cas des tâches dynamiques, nous avons montré que pour toute spécification faisant intervenir l'équité, il était impossible d'implanter un tel détecteur indépendamment de l'algorithme appelé à résoudre le problème.

Par ailleurs, notre étude a montré que dans le cas des tâches statiques, les bornes supérieures pouvaient être considérablement diminuées lorsque l'algorithme était connu du détecteur. De surcroît, certains algorithmes dont la spécification possède une clause d'équité

possèdent un détecteur de défaillances approprié, alors qu'en général, il n'en existe pas pour la spécification qu'ils satisfont.

Une suite naturelle a été de déterminer, pour les tâches statiques, si une borne inférieure existait au surcoût. L'idéal étant d'obtenir un surcoût nul, nous avons cherché à déterminer si des algorithmes sans surcoût et non triviaux pouvaient exister. Nous avons montré qu'une condition sur le code localement exécuté par chaque processeur pouvait être suffisante pour garantir l'auto-stabilisation du système tout entier, indépendamment des hypothèses d'atomicité des actions, de l'ordonnancement du système, et de la topologie du graphe de communication. Du fait que l'algorithme n'est pas modifié, il est forcément sans surcoût dans le cas où aucune défaillance transitoire ne survient dans le système.

En outre, la condition locale que nous avons exhibée permet d'exprimer des solutions à plusieurs problèmes fondamentaux en algorithmique répartie, comme le calcul des plus courts chemins à une source du graphe de communication ou la construction d'un arbre en profondeur.

De manière duale, nous avons développé des outils de synchronisation permettant de construire des algorithmes auto-stabilisants pour des spécifications dynamiques avec un surcoût en mémoire constant, c'est à dire indépendant de la taille du réseau. Ceci rend compte de l'utilité du développement d'algorithmes auto-stabilisants *ad hoc* pour les tâches dynamiques, puisque nous avions montré qu'il n'existait pas de détecteurs pour la plupart des spécifications où intervient l'équité.

Nous avons distingué la synchronisation globale, où tous les processeurs exécutent leurs actions de manière quasi synchrone, de la synchronisation locale, où un processeur est symplement synchronisé avec ses voisins. Un système réparti synchronisé globalement facilite la conception des algorithmes, puisque le non-déterminisme est fortement réduit, au détriment de la rapidité d'exécution, puisque les processeurs les plus rapides doivent attendre les processeurs les plus lents. Au contraire, un système synchronisé localement permet une meilleure concomitance des actions effectuées, au prix d'une conception plus délicate.

Après avoir traité le cas du surcoût de l'auto-stabilisation pour des problèmes statiques et dynamiques, nous avons cherché à déterminer une technique générale pour réduire systématiquement le coût des communications. Pour ce faire, nous nous sommes inspirés de réseaux réels existants, pour lesquels le délai de communication à chaque processeur intermédiaire est borné, et avons donné un cadre général ainsi que des outils d'implantation pour écrire des algorithmes auto-stabilisants dans ce contexte. Nous montrons que pour certaines applications, la quantité de mémoire utilisée par chaque processeur peut être divisée par la taille du système.

Ce dernier point facilite la conception des algorithmes auto-stabilisants, puisque nous avions montré que plus la mémoire transférée entre les processeurs est importante, et plus il est facile de détecter une défaillance transitoire.

Plusieurs des aspects développés dans cette thèse soulèvent de nouveaux problèmes à résoudre.

Multi-Détecteurs Les détecteurs que nous présentons au chapitre 3 sont en mesure de détecter qu'une modification inopinée de la mémoire est survenue dans un système sujet aux défaillances transitoires. Les détecteurs proposés par Chandra et Toueg dans [CT92] sont capables de maintenir la liste des processeurs encore en activité dans un système où des processeurs peuvent tomber en panne définitivement. Une extension naturelle de ces travaux est la contruction d'un oracle réparti qui détecte simultanément ces deux types de défaillances dans un système qui peut être sujet à des perturbations transitoires ou définitives.

Il est facile de voir qu'une simple composition des deux approches est inefficace, puisque les détecteurs de défaillances transitoires peuvent être sujets à des défaillances définitives (et les autres détecteurs répondre vrai dans une configuration incorrecte), et que les détecteurs de défaillances définitives peuvent être sujets à des défaillances transitoires (et donc se tromper sans le savoir concernant la liste des processeurs défaillants).

Langages Auto-stabilisants Le chapitre 4 a montré qu'il était possible de définir une condition sur le programme exécuté localement sur chaque processeur qui garantisse qu'une propriété globale et dynamique (l'auto-stabilisation) était satisfaite. En utilisant la composition équitable présentée dans [Her91], il est évidemment possible de combiner plusieurs programmes locaux satisfaisant cette condition, dont les constantes d'un programme dépendent de la sortie d'un autre. Ceci permet de construire un algorithme auto-stabilisant à partir d'opérateurs de base ayant de plus en plus de connaissances globales sur le système. Toutefois, ceci a pour effet d'augmenter le temps de stabilisation proportionellement au nombre d'algorithmes ainsi superposés.

Une autre approche, celle que nous préconisons, est de combiner ces différents opérateurs en un seul multi-opérateur capable de prendre ses données dans plusieurs ensembles, tout en conservant les propriétés d'auto-stabilisation du système résultant.

En partant de ces multi-opérateurs et de la composition habituelle évoquée plus haut, il est possible de définir un langage de programmation pour les algorithmes répartis. La propriété essentielle de ce langage serait que tout algorithme écrit au moyen de ce langage serait auto-stabilisant sans surcoût. En outre, aucune preuve autre que celle de correction en l'absence de défaillances ne serait nécessaire, réduisant d'autant le temps de développement du système.

Stabilisation Instantanée Une des principales critiques adressées à l'auto-stabilisation est la lenteur de convergence dans le cas où le nombre de défaillances transitoires est limité. Plusieurs approches tiennent compte de ce problème, et notamment la k-stabilisation, qui définit que le temps de stabilisation doit être proportionnel au nombre de fautes effectivement survenues dans le système. Au chapitre 5, nous avons présenté deux algorithmes dont la particularité est d'avoir un temps de stabilisation nul, et ce quel que soit le nombre de fautes. Ces algorithmes, qualifiés d'instantanément stabilisants, sont particulièrement intéressant dans le cas de réseaux capables d'évoluer dynamiquement, ou de réseaux où des

fautes surviennent régulièrement. Les algorithmes instantanément stabilisants connus jusqu'à présent ont trait à la synchronisation des processeurs dans des systèmes asynchrones, mais de nombreux résultats récents semblent indiquer que d'autres problèmes, comme l'exclusion mutuelle locale, admettent eux aussi des solution instantanément stabilisantes.

Une première étude devrait proposer des solutions auto-stabilisantes pour quelques problèmes. Par la suite, et à partir des caractéristiques communes à ces solutions auto-stabilisantes, il apparaît important de caractériser les problèmes qui peuvent admettre une solution instantanément stabilisante.

Communications à Délai Borné Au chapitre 6, nous avons défini une classe d'algorithmes auto-stabilisants pour un modèle de communication où tous les processeurs retransmettent les messages sur le réseau avec un délai fixé à l'avance et indépendant de la taille du réseau. Le routage que nous proposons dans les réseaux eulériens est simple mais présente l'inconvénient de parcourir tous les liens, même quand un chemin plus court existe.

Une extension de ce travail consiste à relâcher la contrainte du délai borné sur certains processeurs du système, de manière d'une part à offrir un routage plus efficace (ces processeurs pourraient rechercher un chemin plus rapide à partir de l'adresse de destination stockée dans le message), et d'autre part à supporter un plus grand nombre de topologies (un processeur possédant plus de canaux entrants que sortants pourrait stocker un ou plusieurs messages en attendant de pouvoir les retransmettre) ainsi que des débits variables (un processeur dont le débit d'entrée est différent du débit de sortie pourrait ainsi compenser l'arrivée trop rapide des informations).

Applications de l'Auto-stabilisation Le manque d'applications "réelles" à l'auto-stabilisation est principalement dû au fait que les algorithmes sont écrits à un niveau d'abstraction assez élevé et en faisant l'hypothèse que l'algorithme est exécuté à partir d'un système d'exploitation également stabilisant. Nos travaux présentent plusieurs points sur lesquels cette hypothèse apparait vérifiée.

Au chapitre 4, les opérateurs associatifs que nous avons utilisés sont implantés de manière efficace sur une machine parallèle existante : la Maille Associative, développée à l'Institut d'Electronique Fondamentale, à Orsay. Ceci augure d'une mise en œuvre possible des algorithmes auto-stabilisants proposés au chapitre 4.

Par ailleurs, le chapitre 6 place au même niveau le protocole de communication et l'application auto-stabilisante qui l'utilise. Cette approche pose les bases de systèmes d'exploitation auto-stabilisants en définissant les couches système de plus bas niveau. Une fois ces fondations effectivement implantées dans des réseaux réels, de nouvelles couches plus élaborées peuvent venir se greffer.

Bibliographie

[AAEG93] A Arora, P Attie, M Evangelist, and M G Gouda. Convergence of iteration systems. *Distributed Computing*, 7 :43–53, 1993.

[AB93] Y Afek and G M Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7 :27–34, 1993.

[ABC+98] G Alari, J Beauquier, J Chacko, A K Datta, and S Tixeuil. A fault-tolerant distributed sorting algorithm on tree networks. In *Proceedings of the Seventeenth IEEE International Performance, Computing, and Communications Conference (IPCCC'98)*, 1998.

[ABDT98] L O Alima, J Beauquier, A K Datta, and S Tixeuil. Self-stabilization with global rooted synchronizers. In *Proceedings of the Eighteenth International Conference on Distributed Computing Systems (ICDCS'98)*, 1998.

[AD97] Y Afek and S Dolev. Local stabilizer. In *Proceedings of the Sixteenth Annual ACM SYmposium on Principles of Distributed Computing*, page 287, 1997.

[ADG91] A Arora, S Dolev, and M G Gouda. Maintaining digital clocks in step. *Parallel Processing Letters*, 1 :11–18, 1991.

[AG94] A Arora and M G Gouda. Distributed reset. *IEEE Transactions on Computers*, 43 :1026–1038, 1994.

[AH93] E Anagnostou and V Hadzilacos. Tolerating transient and permanent failures. In *Proceedings of the Seventh International Workshop on Distributed Algorithms (WDAG'93), LNCS 725*, pages 174–188, September 1993.

[AHOvdP94] D Alstein, J H Hoepman, B E Olivier, and P I A van der Put. Self-stabilizing mutual exclusion on directed graphs. Technical Report CS-R9513, CWI, 1994. Published in Computer Science in the Netherlands (CSN 94), pages 45–53.

[AKM+93] B Awerbuch, S Kutten, Y Mansour, B Patt-Shamir, and G Varghese. Time optimal self-stabilizing synchronization. In *Proceedings of STOC'93*, pages 652–661, 1993.

[AKY90] Y Afek, S Kutten, and M Yung. Memory efficient self-stabilizing protocols for general networks. In *Proceedings of the Fourth International Workshop on Distributed Algorithms (WDAG'90), LNCS 486*, 1990.

[APSV91] B Awerbuch, B Patt-Shamir, and G Varghese. Self-stabilization by local checking and correction. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.

[APSVD94] B Awerbuch, B Patt-Shamir, G Varghese, and S Dolev. Self-stabilizing by local checking and global reset. In *Distributed Algorithms Eigth International Workshop Proceedings, Springer-Verlag LNCS :857*, pages 326–339, 1994.

[Arn92] A Arnold. *Systèmes de Transitions Finis et Sémantique des Processus Communicants*. Masson, 1992.

[AW98] H Attiya and J Welch. *Distributed Computing : Fundamentals, Simulations and Advanced Topics*. The McGraw-Hill Companies, 1998.

[Awe85] B Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4) :804–823, 1985.

[Bar96] V C Barbosa. *An Introduction to Distributed Algorithms*. The MIT Press, 1996.

[BBF99] J Beauquier, B Bérard, and L Fribourg. A new rewrite method for proving convergence of self-stabilizing systems. In *Proceedings of the Thirteenth International Symposium on Distributed Computing (DISC'99)*, pages 240–253, September 1999.

[BCBT96] A Basu, B Charron-Bost, and S Toueg. Simulating reliable links in the presence of process crashes. In *Proceedings of the Tenth International Workshop on Distributed Algorithms (WDAG'96), LNCS 1151*, 1996.

[BCD95] J Beauquier, S Cordier, and S Delaët. Optimum probabilistic self-stabilization on uniform rings. In *Proceedings of the Second Workshop on Self-stabilizing Systems*, May 1995.

[BD93] J Beauquier and Delaët. Classes of self-stabilizing protocols. In *Proceedings of the Future Trends of Distributed Computing*, pages 361–365, September 1993.

[BDDT98] J Beauquier, S Delaët, S Dolev, and S Tixeuil. Transient failure detectors. In *Proceedings of the Twelfth International Conference on Distributed Computing (DISC'98)*, 1998.

[BDPV98] A Bui, A K Datta, F Petit, and V Villain. Space optimal and fast self-stabilizing pif in tree networks. Technical Report RR 98-06, LaRIA, University of Picadie Jules Verne, 1998.

[BDPV99] A Bui, A K Datta, F Petit, and V Villain. State-optimal snap-stabilizing pif in tree networks. In *Proceedings of the Fourth Workshop on Self-stabilizing Systems*, pages 78–85, 1999.

[BDT99] J Beauquier, A K Datta, and S Tixeuil. Self-stabilizing census with cut-through constraints. In *Proceedings of the Fourth Workshop on Self-stabilizing Systems (WSS'99)*, pages 70–77, 1999.

[BGM93] J E Burns, M G Gouda, and R E Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7 :35–42, 1993.

[BKM97] J Beauquier and S Kekkonen-Moneta. On ftss-solvable distributed problems. In *Proceedings of the Sixteenth International Symposium on Principles of Distributed Computing (PODC'97)*, August 1997.

[BKT99] J Beauquier, S Kutten, and S Tixeuil. Self-stabilization in eulerian networks with cut-through constraints. Technical Report 1200, LRI, Université de Paris Sud, 1999.

[BSW69] K A Bartlett, R A Scantlebury, and P T Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5) :260–261, May 1969.

[CFG92] J-M Couvreur, N Francez, and M G Gouda. Asynchronous unison. In *ICDCS92 Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 486–493, 1992.

[CS94] S Chandrasekar and P K Srimani. A self-stabilizing algorithm to synchronize digital clocks in a distributed system. *Computers and Electrical Engineering*, 20(6) :439–444, 1994.

[CT92] T D Chandra and S Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 325–340, 1992.

[CV99] A Costello and G Varghese. The FDDI MAC meets self-stabilization. pages 1–9, 1999.

[CYH91] N S Chen, H P Yu, and S T Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39 :147–151, 1991.

[DDT99] S K Das, A K Datta, and S Tixeuil. Self-stabilizing algorithms in dag structured networks. In *International Symposium on Parallel Architectures and Networks (I-SPAN'99)*, pages 190–195, 1999. To appear in Parallel Processing Letters.

[Del95] S Delaët. *Auto-stabilisation : Modèle et Applications à l'Exclusion Mutuelle.* PhD thesis, December 1995.

[DF97] C Delporte and H Fauconnier. Un algorithme de consensus résistant aux défaillances définitives de processus et aux pertes de messages. In *Proceedings of the Ninth Rencontres Francophones du Parallélisme (Renpar'97)*, 1997.

[DH97] S Dolev and T Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997.

[Dij74] E W Dijkstra. Self-stabilization in spite of distributed control. *Communications of the ACM*, 17(11) :643–644, November 1974.

[DIM93] S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7 :3–16, 1993.

[DIM97] S Dolev, A Israeli, and S Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4) :424–440, 1997.

[Dol97] S Dolev. Possible and impossible self-stabilizing digital clock synchronization in general graphs. *Journal of Real-Time Systems*, 12(1) :95–107, 1997.

[dR94] J de Rumeur. *Communication dans les Réseaux de Processeurs*. Masson, 1994. Ouvrage Collectif.

[DT98a] S Delaët and S Tixeuil. Un algorithme auto-stabilisant en dépit de communications non fiables. *Technique et Science Informatiques*, 5(17), 1998.

[DT98b] B Ducourthial and S Tixeuil. Self-stabilizing global computations with r-operators. In *Proceedings of the Second International Conference on Principles of Distributed Computing (OPODIS'98)*, pages 99–113, 1998.

[DT99] B Ducourthial and S Tixeuil. Self-stabilization with path algebra. Technical Report 1202, LRI, Université de Paris Sud, 1999. To appear in Theoretical Computer Science.

[Duc99] B Ducourthial. *Les Réseaux Associatifs, un modèle de programmation à Parallélisme de Données, pour Algorithmes et Données irréguliers, à Primitives de Calcul Asynchrones*. PhD thesis, 1999.

[DW95] S Dolev and J L Welch. Self-stabilizing clock synchronization in the presence of byzantine faults. In *Proceedings of the Second Workshop on Self-stabilizing Systems*, pages 9.1–9.12, 1995.

[FLP85] M J Fisher, N A Lynch, and M S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2) :374–382, April 1985.

[GGHP96] S Ghosh, A Gupta, T Herman, and S V Pemmaraju. Fault-containing self-stabilizing algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 45–54, 1996.

[GH97] M G Gouda and F Haddix. The linear alternator. In *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pages 31–47. Carleton University Press, 1997.

[GK93] S Ghosh and M H Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing*, 7 :55–59, 1993.

[Her91] T Herman. *Adaptativity through Distributed Convergence*. PhD thesis, August 1991.

[HG95] T Herman and S Ghosh. Stabilizing phase-clocks. *Information Processing Letters*, 54 :259–265, 1995.

[IJ90] A Israeli and M Jalfon. Self-stabilizing extensions for message passing systems. In *Proceedings of the Fourth International Workshop on Distributed Algorithms (WDAG'90), LNCS 486*, 1990.

[JADT99] C Johnen, L O Alima, A K Datta, and S Tixeuil. Self-stabilizing neighborhood synchronizer in tree networks. In *Proceedings of the Nineteenth International Conference on Distributed Computing Systems (ICDCS'99)*, pages 487–494, 1999.

[JàJ92] J JàJà. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, 1992.

[Joh97] C Johnen. Memory-efficient self-stabilizing algorithm to construct BFS spanning trees. In *Proceedings of the WSS'97*, pages 125–140. Carleton University Press, 1997.

[KM98] S Kekkonen-Moneta. *Auto-stabilisation et Tolérance aux Fautes*. PhD thesis, 1998.

[KP93] S Katz and K J Perry. Self-stabilizing extensions for message passing systems. *Distributed Computing*, 7(1) :17–26, 1993.

[KPS97] S Kutten and B Patt-Shamir. Time-adaptive self-stabilization. In *Proceedings of the Sixteenth Annual ACM SYmposium on Principles of Distributed Computing*, pages 149–158, 1997.

[Kru79] H S M Kruijer. Self-stabilization (in spite of distributed control) in tree-structured systems. *Information Processing Letters*, 8 :91–95, 1979.

[Lam94] L Lamport. Solved problems, unsolved problems and non-problems in concurrency. In *Proceedings of the Third ACM SYmposium on Principles on Distributed Computing*, pages 1–11, 1994.

[Lav95] C Lavault. *Evaluation des Algorithmes Distribués*. Hermes, 1995.

[LS92] C Lin and J Simon. Observing self-stabilization. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, pages 113–123, 1992.

[Lyn96] N A Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[Mas95] T Masuzawa. A fault-tolerant and self-stabilizing algorithm for the topology problem. In *Proceedings of the Second Workshop on Self-stabilizing Systems*, pages 1.1–1.15, May 1995.

[Pet98] F Petit. *Efficacité et Simplicité dans les Algorithmes Distribués Auto-stabilisants de Parcours en Profondeur de Jeton*. PhD thesis, January 1998.

[RH90] M Raynal and J-M Helary. *Synchronization and Control of Distributed Systems and Programs*. John Wiley and Sons, Chichester, UK, 1990.

[SG89] J M Spinelli and R G Gallager. Event driven topology broadcast without sequence numbers. *IEEE Transaction on Communications*, 37(5) :468–474, May 1989.

[Ste76] N V Stenning. A data transfer protocol. *Computer Networks*, 1(2) :99–110, September 1976.

[TB96] S Tixeuil and J Beauquier. Self-stabilizing token ring. In *Proceedings of the Eleventh International Conference on System Engineering (ICSE'96)*, 1996.

[Tch81] M Tchuente. Sur l'auto-stabilisation dans un réseau d'ordinateurs. *RAIRO Informatique Theoretique*, 15 :47–66, 1981.

[Tel94] G Tel. *Introduction to Distributed Algorithms.* Cambridge university press, 1994.

[Var94] G Varghese. Self-stabilization by counter flushing. In *PODC94 Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 244–253, 1994.

[Wal91] J Walrand. *Communication Networks : a First Course.* Aksen Associate, 1991.

[WZ89] D-W Wang and L D Zuck. Tight bounds for the sequence transmission problem. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing (PODC'89)*, pages 73–83, August 1989.

Résumé.

Quand un système réparti est sujet à des défaillances transitoires qui modifient arbitrairement son état, il est crucial de pouvoir retrouver un comportement correct au bout d'un temps fini. L'*auto-stabilisation* présente une telle garantie, mais en général au prix de ressources importantes. Dans cette thèse, notre démarche a consisté à minimiser ces ressources lorsque cela était possible.

Nous avons développé le concept de *détecteur de défaillances transitoires*, des oracles appelés par les processeurs du système, qui indiquent si des défaillances transitoires sont survenues, en un temps constant. Notre implantation permet de classifier les problèmes classiques suivant les ressources spécifiques nécessaires à la détection d'une erreur. Pour les tâches statiques, une suite naturelle a été de montrer qu'une condition sur le code localement exécuté par chaque processeur pouvait être suffisante pour garantir l'auto-stabilisation du système tout entier, indépendamment des hypothèses d'exécution et de la topologie du graphe de communication. Du fait que l'algorithme n'est pas modifié, il est forcément *sans surcoût*. De manière duale, nous avons développé des outils de *synchronisation* permettant de construire des algorithmes auto-stabilisants pour des spécifications dynamiques avec un surcoût en mémoire constant, c'est à dire indépendant de la taille du réseau. En outre, l'un des algorithmes présentés est *instantanément stabilisant*. Enfin, nous avons présenté une technique générale pour réduire systématiquement le coût des communications, en garantissant un *délai de retransmission borné*, et nous avons donné un cadre général ainsi que des outils d'implantation pour écrire des algorithmes auto-stabilisants dans ce contexte.

Efficient Self-stabilization.

When a distributed system is subject to transient failures that arbitrarily modify its state, it is crucial to recover a correct behavior within finite time. *Self-stabilization* offers such a guarantee, but usually uses large ressources. In this thesis, we focus on minimizing these ressources when such solutions exist.

We introduced the concept of *transient failure detectors*, oracles that are called by processors, which notify if transient failures occurred within constant time. Our implementation enables classifying classic problems according to the specific ressources dedicated to error detection. A natural extension was to show that a local property on the local code executed by each processor is sufficient to guarantee self-stabilization of the whole system, whatever the computation assumptions and communication graph may be. Since the original algorithm is not modified, it is *overhead-free*. Similarly, we developped two *synchronizers*, that enable dynamic tasks to be solved self-stabilizingly, whose memory overhead is constant. Moreover, one of them is *snap-stabilizing*. Finally, we presented a general technique to systematically reduce the communication cost, assuming a *bounded retransmission delay*, and we gave a general framework and tools to design self-stabilizing algorithms in this context.

DISCIPLINE: Informatique
SPÉCIALITÉ: Auto-stabilisation
MOTS-CLÉS: Systèmes répartis, complexité, auto-stabilisation.

Laboratoire de Recherche en Informatique, Université Paris Sud, 91405 Orsay cedex, France