

UNIVERSITE PARIS DAUPHINE
U.F.R. SCIENCES DES ORGANISATIONS



N° attribué par la bibliothèque

| | | | | | | | | | |
|--|--|--|--|--|--|--|--|--|--|
| | | | | | | | | | |
|--|--|--|--|--|--|--|--|--|--|

THESE

Pour l'obtention du titre de
DOCTEUR EN INFORMATIQUE

spécialité : systèmes répartis

***Auto-stabilisation : gestion de robots mobiles et
confinement de fautes***

Candidate : Joyce EL HADDAD

JURY

Directeur de thèse : Serge HADDAD
Professeur à l'Université Paris Dauphine

Rapporteurs : Alain BUI
Professeur à l'Université de Reims Champagne-Ardenne
Hugues FAUCONNIER
Maître de conférences à l'Université Paris VII

Suffrageants : Béatrice BERARD
Professeur à l'Université Paris Dauphine
Colette JOHNEN
Maître de conférences à l'Université Paris XI
Pierre SENS
Professeur à l'Université Paris VI

Présentée et soutenue publiquement le 08 Décembre 2004

L'université n'entend donner aucune approbation ni improbation aux opinions émises dans les thèses : ces opinions doivent être considérées comme propres à leurs auteurs.

Table des matières

| | |
|--|-----------|
| Introduction | 1 |
| 1 Systèmes répartis | 5 |
| 1.1 Introduction | 6 |
| 1.1.1 Objectifs des systèmes répartis | 6 |
| 1.1.2 Spécificités des systèmes répartis | 7 |
| 1.1.3 Problèmes classiques en algorithmique répartie | 9 |
| 1.2 Hypothèses de modélisation | 12 |
| 1.2.1 Modes de communication | 12 |
| 1.2.2 Topologies classiques | 14 |
| 1.3 Tolérance aux défaillances | 15 |
| 1.3.1 La tolérance aux pannes | 16 |
| 1.3.2 L’auto-stabilisation | 17 |
| 1.3.3 L’auto-stabilisation améliorée | 23 |
| 1.4 Positionnement de notre travail | 27 |
| 2 Modèle | 29 |
| 2.1 Introduction | 30 |
| 2.2 Notions préliminaires | 30 |
| 2.2.1 Graphes | 30 |
| 2.2.2 Systèmes de transitions | 32 |
| 2.3 Composants de base d’un système réparti | 35 |
| 2.3.1 Liens de communication | 35 |
| 2.3.2 Processeurs | 38 |
| 2.4 Systèmes répartis | 40 |
| 2.4.1 Exécutions | 41 |
| 2.4.2 Atomicité | 43 |
| 2.4.3 Propriété d’équité | 44 |
| 2.4.4 Démon | 44 |
| 2.5 Auto-stabilisation | 47 |

| | | |
|----------|--|-----------|
| 2.5.1 | Correction | 47 |
| 2.5.2 | Convergence | 48 |
| 2.5.3 | Clôture | 50 |
| 2.6 | Mesures de complexité | 52 |
| 2.7 | Conclusion | 54 |
| 3 | Coopération auto-stabilisante | 57 |
| 3.1 | Introduction | 58 |
| 3.2 | Ordonnancement des déplacements des robots | 60 |
| 3.2.1 | Les hypothèses du modèle | 61 |
| 3.2.2 | Algorithme d'ordonnancement non auto-stabilisant | 63 |
| 3.3 | Preuve de correction | 64 |
| 3.4 | Ordonnancement auto-stabilisant | 67 |
| 3.4.1 | Les hypothèses du modèle | 67 |
| 3.4.2 | Algorithme d'ordonnancement auto-stabilisant | 68 |
| 3.4.3 | Idée de la preuve | 69 |
| 3.5 | Preuve formelle | 73 |
| 3.5.1 | Sémantique probabiliste | 73 |
| 3.5.2 | États stables | 76 |
| 3.5.3 | D'un état initial à un état stable | 78 |
| 3.6 | Conclusion | 79 |
| 4 | Routage auto-stabilisant | 81 |
| 4.1 | Introduction | 82 |
| 4.2 | Routage auto-stabilisant dans les systèmes de robots mobiles | 83 |
| 4.2.1 | Le modèle | 83 |
| 4.2.2 | Algorithme | 86 |
| 4.3 | Preuve de stabilisation | 87 |
| 4.4 | Conclusion | 92 |
| 5 | Confinement de fautes dans le problème d'arbre couvrant | 93 |
| 5.1 | Introduction | 94 |
| 5.2 | Construction d'un arbre couvrant | 95 |
| 5.2.1 | Les hypothèses du modèle | 95 |
| 5.2.2 | Algorithme initial | 97 |
| 5.3 | Confinement de fautes dans la construction d'un arbre couvrant | 99 |
| 5.3.1 | Description informelle | 100 |
| 5.3.2 | Algorithme | 104 |
| 5.4 | Preuve de la maîtrise d'une faute unique | 105 |

| | | |
|-------|---|------------|
| 5.5 | Preuve de correction partielle | 121 |
| 5.6 | Preuve de l'auto-stabilisation | 122 |
| 5.6.1 | Disparition des fausses identités | 123 |
| 5.6.2 | Maintien d'une requête ou couverture de l'arbre | 125 |
| 5.6.3 | Couverture de <i>Arbre</i> | 131 |
| 5.7 | Conclusion | 135 |
| | Conclusion | 137 |

Introduction

“Le temps que vous lisiez ces lignes, sept cents millions de fourmis seront nées sur la planète. Sept cents millions d’individus dans une communauté estimée à un milliard de milliards, et qui a ses villes, sa hiérarchie, ses colonies, son langage, sa production industrielle, ses esclaves, ses mercenaires...”

Werber, 1991

Les insectes sociaux comme les fourmis présentent au niveau collectif des capacités supérieures à la somme des capacités individuelles. Cette performance est le résultat des interactions directes ou indirectes entre les individus. En effet, les fourmis disposent d’une “intelligence collective”, quand elles sont à la recherche de nourriture ou qu’elles combattent un prédateur. Chacune dans la colonie effectue des actions très simples, mais réunies, elles sont capables de choses beaucoup plus complexes, comme découvrir le chemin le plus court entre la fourmilière et une source de nourriture.

La quête de la nourriture. Supposons qu’une colonie de fourmis n’ait aucune information sur l’emplacement de nourriture dans son environnement. Les fourmis cherchent tout d’abord la nourriture en marchant au hasard, de façon similaire à une équipe de sauveteurs humains ratissant une zone pour retrouver un disparu. Cependant, deux problèmes les empêchent d’appliquer directement la technique des “équipes de secours”. Tout d’abord, comment une fourmi, une fois la nourriture découverte, retrouve-t-elle la route du nid ? Ensuite, même si la fourmi retrouve le chemin du nid, comment peut-elle informer les autres fourmis de l’emplacement de la nourriture ?

Le plus court chemin. Les réponses résident dans un usage intelligent des phéromones (une “odeur chimique” que les fourmis utilisent pour communiquer avec leurs congénères). Pour résoudre le problème de la localisation du nid, les fourmis laissent derrière elles une trace de phéromones quand elles recherchent de la nourriture (comme le Petit Poucet semant des cailloux blancs dans les bois pour retrouver le chemin de la maison). Ainsi, quand une fourmi trouve de la nourriture, elle peut suivre sa propre trace de phéromones jusqu’au nid. Sur le chemin du retour, la fourmi résout le deuxième problème, en redépo-

sant des phéromones, créant ainsi une piste avec une odeur plus forte. Quand les autres fourmis tombent sur une telle trace de phéromones, elles abandonnent leurs recherches, et se mettent à suivre la piste, ajoutant une couche supplémentaire de phéromones. Cette méthode permet également d'atteindre un objectif inattendu au départ : trouver le plus court chemin qui mène à la nourriture. En effet, le chemin le plus court est parcouru avec une fréquence plus rapide par les fourmis qui l'arpentent. Au fur et à mesure, la quantité de phéromones sur la piste la plus courte va donc se renforcer. Les autres fourmis préférant suivre la trace la plus odorante, elles emprunteront le plus court chemin, confortant ainsi le précepte : *“Il ne faut pas penser à l'objectif à atteindre, il faut seulement penser à avancer. C'est ainsi, à force d'avancer, qu'on atteint ou qu'on double ses objectifs sans même sans apercevoir”* (Werber, 1995).

Le bâton dans les roues. Même par jours de grand vent, les fourmis doivent ramener de la nourriture pour la colonie. Pourtant, lors de telles journées, il se peut fort qu'une branche arrachée vienne malencontreusement couper le chemin vers la nourriture. La tâche des fourmis ne devant pas connaître relâche, elles contournent l'obstacle, certaines passant par la gauche et d'autres par la droite.

Toujours le plus court chemin. La “reconnexion” à l'autre partie de la trace se fait donc aléatoirement par la gauche ou par la droite, conduisant ainsi à deux pistes possibles vers la nourriture. Néanmoins, les fourmis empruntant le détour le plus court réaliseront plus d'allers-retours que celles empruntant la direction opposée. Au fur et à mesure, les fourmis renforceront la piste la plus courte et l'autre disparaîtra. De cette manière, les fourmis empruntent toujours le plus court chemin.

Des colonies de fourmis aux réseaux d'ordinateurs. Les fourmis expérimentent ainsi dans la nature le phénomène de convergence qu'on retrouve comme propriété des algorithmes auto-stabilisants pour les systèmes répartis en informatique (tel que le réseau Internet par exemple). Dans de tels systèmes, nous n'avons pas des fourmis sortant de leur nid à la recherche de la nourriture, mais par exemple des messages transitant d'un expéditeur à un destinataire. Les règles régissant le comportement des fourmis sont alors transposables aux processeurs situés en chacun des nœuds d'un réseau informatique, et conduisent à un algorithme auto-stabilisant de routage des messages. En effet, un message peut être vu comme une fourmi virtuelle transitant sur le réseau et marquant les nœuds empruntés. Régulièrement, un nœud va requérir de ses voisins leurs distances présumées au destinataire (de façon similaire à des fourmis virtuelles qui partiraient chacune dans leur propre direction pour recueillir de l'information). Pour ce faire, chacun de ses voisins va lui envoyer un message comportant cette information. Lorsque les

messages envoyés par ses voisins (l'information recueillie par les fourmis virtuelles auprès des voisins) arrivent sur le nœud, c'est le message comportant la distance présumée au destinataire la plus courte qui est retenue (le marquage en phéromone virtuelle étant d'autant plus intense que le chemin parcouru est court). En procédant ainsi, le marquage de chaque nœud va se stabiliser sur la valeur du plus court chemin pour atteindre le destinataire. Pour un message cherchant à atteindre un destinataire donné, une stratégie simple consiste alors à parcourir le réseau dans la direction indiquée par les marquages les plus intenses, exactement comme les fourmis suivant la piste de plus forte "odeur". En cas de panne d'un lien (comme une branche venant interrompre la piste des fourmis), et en fonction de l'ordre de réception des informations des voisins, les messages peuvent être acheminés un temps via des chemins détournés (comme les fourmis empruntant le détour le plus long), mais ils finiront toujours par être acheminés via le plus court chemin vers le destinataire (comme les fourmis "reconnectant" les deux parties d'une même trace par le chemin le plus court).

Objectif de la thèse. L'objectif de cette thèse est précisément l'étude d'algorithmes répartis tolérant des pannes dites transitoires. Nos travaux portent d'une part sur l'importation de l'approche auto-stabilisante (qui permet de recouvrer les fonctions d'un système à partir d'un état arbitraire issu d'une panne transitoire) dans un environnement mobile constitué de robots autonomes communiquant par intermittence. En effet, dans un tel contexte, le protocole de synchronisation est particulièrement délicat à concevoir dès lors que les robots sont sujets à des pannes temporaires. Nos travaux portent d'autre part sur l'intégration de la propriété de confinement de fautes dans un algorithme auto-stabilisant de construction d'un arbre couvrant.

Organisation de la thèse. Les deux premiers chapitres sont consacrés à la présentation des systèmes répartis, à leur modélisation, et aux méthodes de démonstration des algorithmes utilisés dans ce contexte. Dans le chapitre 1, nous introduisons le domaine des systèmes répartis. Nous rappelons certaines de leurs propriétés caractéristiques, et présentons quelques problèmes classiques représentatifs des travaux menés en algorithmique répartie. Nous nous intéressons également aux différentes techniques de tolérance aux pannes, en mettant un accent particulier sur l'auto-stabilisation et ses différentes variantes. Le chapitre 2 est consacré à la modélisation des concepts présentés dans le chapitre précédent et à la présentation de quelques notions fondamentales en théorie des graphes et dans les systèmes de transitions. Nous introduisons des outils formels permettant de décrire le comportement d'un système réparti au cours du temps. Nous présentons également certaines mesures de complexité utilisées pour comparer les algorithmes auto-stabilisants. Les chapitres 3 et 4 portent sur l'importation de l'approche auto-stabilisante

(qui permet de recouvrer les fonctions d'un système à partir d'un état arbitraire issu d'une panne transitoire) dans un environnement mobile constitué de robots autonomes communiquant par intermittence. Dans de tels systèmes, deux types d'algorithmes de communications sont nécessaires : l'algorithme de synchronisation entre deux robots voisins afin d'établir une communication point-à-point (temporaire), et l'algorithme de routage permettant l'échange de données entre deux robots (même s'ils sont distants). Le premier algorithme fait l'objet du chapitre 3, dans lequel nous utilisons les réseaux de Petri comme outil pour prouver la correction d'un algorithme d'ordonnement des déplacements des robots. Nous décrivons ensuite la transformation de cet algorithme en un algorithme auto-stabilisant. Le deuxième algorithme, quant à lui, fait l'objet du chapitre 4. Enfin, le chapitre 5 porte sur l'intégration de la propriété de confinement de fautes dans un algorithme auto-stabilisant de construction d'un arbre couvrant. La principale caractéristique de l'algorithme proposé est que seuls le site fautif et ses voisins sont susceptibles d'agir pour rétablir un état correct du système après une faute.

Chapitre 1

Systemes répartis

Résumé. *Dans ce chapitre, nous commençons par présenter le domaine des systèmes répartis. Nous rappelons certaines de leurs propriétés caractéristiques, et présentons quelques problèmes classiques représentatifs des travaux menés dans les algorithmes répartis. Suite à cette introduction portant principalement sur les enjeux et les caractéristiques des systèmes répartis, nous revenons sur les hypothèses de modélisation, en décrivant plus précisément comment fonctionnent de tels systèmes. Nous nous intéressons ensuite aux différentes techniques de tolérance aux pannes, en distinguant l'approche classique de l'approche par auto-stabilisation. Nous détaillons certains des avantages et des inconvénients de l'auto-stabilisation. Ces derniers justifient le raffinement de cette approche de façon à traiter les fautes plus localement et à les corriger plus rapidement. C'est précisément l'objet des algorithmes k -stabilisants, proportionnels ou confinant les fautes, que nous présentons par la suite.*

1.1 Introduction

Le domaine des systèmes répartis est vaste et assez récent. Ces systèmes répondent à la demande croissante de puissance de calcul et de communication. Les avantages des systèmes répartis les ont rendus indispensables. La répartition et la coopération sont les deux principaux concepts à la base des systèmes répartis. La répartition concerne aussi bien le traitement des données que leur stockage. La coopération, quant à elle, signifie le dialogue entre des applications ayant des rôles complémentaires pour la réalisation des tâches.

Formellement, un *système réparti* est un ensemble d'entités autonomes (*sites*, *processeurs* ou *processus*) interconnectés par des liens de communication (aussi appelés *canaux* ou *registres*) :

- ces entités sont *autonomes* : elles sont capables d'effectuer des calculs indépendamment les unes des autres ;
- elles sont *interconnectées* : il leur est possible d'échanger des informations par l'intermédiaire des liens de communication.

Le modèle d'interconnexion des sites et des liens de communication définit ce qu'on appelle la *topologie* du système. Cette topologie peut être modélisée par un *graphe de communication*. Dans ce formalisme, les entités du système sont appelées les *nœuds* et on utilise des *arêtes* pour représenter les liens de communications entre ces entités. Un exemple de graphe de communication d'un système réparti est donné sur la figure 1.1.

1.1.1 Objectifs des systèmes répartis

Les systèmes répartis se sont répandus dans le monde informatique avec l'avènement des réseaux haut-débit, qu'ils soit locaux (i.e., Intranet, Ethernet) ou longue distance (i.e., Extranet, Internet), auxquels sont connectés un nombre variable d'ordinateurs. Ils présentent certaines caractéristiques propres, les distinguant des systèmes centralisés, qui rendent leur utilisation parfois préférable, voire incontournable :

- **L'échange d'information** : le besoin d'échanger des informations entre ordinateurs s'est développé dans les années soixante, alors que les universités et les principales entreprises venaient de s'équiper d'ordinateurs centraux. La coopération entre ces différentes organisations a été facilitée par l'échange de données entre ordinateurs, ce qui a encouragé le développement des réseaux longue distance. Aujourd'hui, ces réseaux sont partout et permettent aux particuliers comme aux professionnels d'échanger des données et de partager l'information.
- **Le partage de ressources** : bien que le coût des ordinateurs individuels ait considérablement baissé, il n'en est pas de même pour les périphériques (comme les

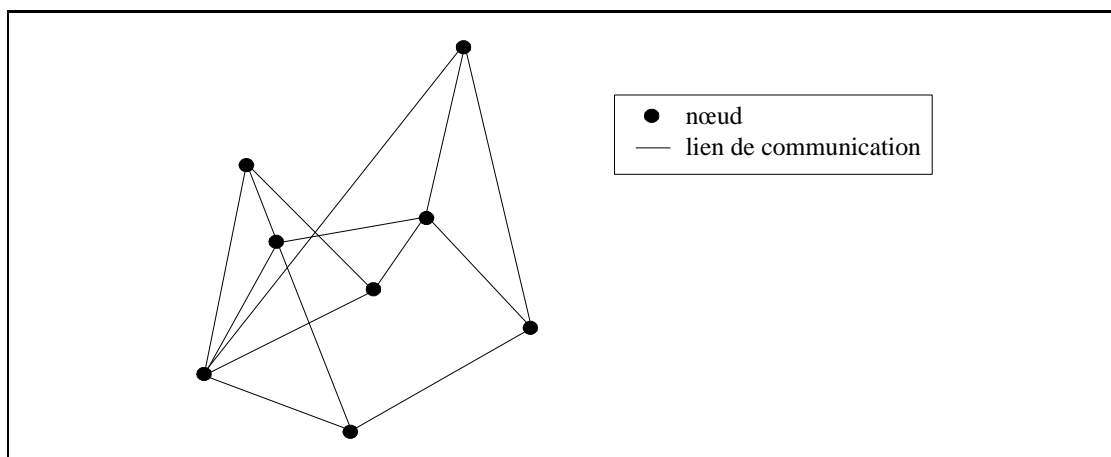


FIG. 1.1 – Le graphe associé à un système distribué

imprimantes, les scanners ou encore les unités de sauvegarde). Le développement des réseaux locaux a permis aux ordinateurs les moins coûteux d'accéder à ces périphériques partagés. Cette approche diminue les coûts par rapport aux systèmes centralisés et accroît l'extensibilité par l'ajout progressif de composants.

- **La fiabilité** : un système réparti, constitué de plusieurs composants autonomes, peut continuer à fonctionner correctement en cas de défaillance d'une partie de ces composants. En effet, la partie du système qui continue à fonctionner correctement peut prendre en charge les travaux assignés aux composants défaillants, ou encore rétablir les composants défaillants dans un état à partir duquel le service pourra de nouveau être assuré. En raison de la redondance des ressources, les systèmes répartis offrent naturellement un certain degré de tolérance aux fautes qui les rendent potentiellement plus fiables que les systèmes centralisés. La duplication des informations sur différents sites est une technique de tolérance aux fautes qui réduit considérablement les risques de perte. Par ailleurs, l'exécution d'une même application critique sur plusieurs sites peut pallier à la défaillance d'un certain nombre d'entre eux.
- **Les performances** : le fait de disposer de plusieurs unités de calcul offre la possibilité de diviser une tâche en plusieurs sous-tâches et de les exécuter simultanément. Ainsi, le temps de calcul peut devenir inversement proportionnel aux ressources qui lui sont consacrées.

1.1.2 Spécificités des systèmes répartis

Tout en offrant des nouvelles possibilités de développement d'applications informatiques, les systèmes répartis engendrent divers problèmes propres, découlant de leurs prin-

principales caractéristiques. Des algorithmes spécifiques, adaptés au contexte réparti, ont été conçus : *les algorithmes répartis*. Les différences avec les algorithmes centralisés tiennent en trois points essentiels (Lavallé, 1990; Raynal, 1991, 1992a,b; Tel, 1994; Lynch, 1996) :

- **Ignorance de l'état global** : à tout instant, dans un système *centralisé*, le processeur exécutant un protocole connaît l'*état global* du système. Dans un système *réparti*, chaque processeur 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 autres processeurs, et se baser sur ces informations pour prendre des décisions. 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.
- **Absence d'horloge globale** : dans un système réparti, chaque processeur possède sa propre horloge. Ces différentes horloges peuvent ne pas être synchronisées. Lamport (1978) a proposé de définir pour ce type de systèmes une notion de temps logique permettant de comparer des événements du point de vue de leur ordre d'exécution : d'une part, sur un site, les événements locaux peuvent être ordonnés en se basant sur l'ordre de leur exécution et d'autre part l'émission d'un message sur le site émetteur précède toujours sa réception sur le site récepteur. Cela correspond à la notion de précedence causale. Cette relation définit un ordre partiel sur les événements du système. Des événements e et e' non comparables sont dits concurrents. A titre d'exemple, considérons un système composé de trois sites $S1$, $S2$ et $S3$ et de l'ensemble des événements $\{e11, e12, e13, e14, e21, e22, e23, e31, e32, e33\}$ correspondant à des émissions et à des réceptions de messages et où eij représente le $j^{ième}$ événement sur le site Si . La figure 1.2 représente l'ensemble des événements du système ainsi que la relation de dépendance causale correspondante. Ainsi, l'événement $e22$ est précédé directement par l'événement $e21$ puisqu'ils se produisent sur le même site $S2$, et par l'événement $e32$ puisqu'il représente l'émission du message dont $e22$ figure la réception. De même, l'événement $e32$ est précédé directement par $e31$, lui-même précédé par $e11$. En procédant ainsi, on trouve l'ensemble $\{e21, e32, e31, e11\}$ des événements antérieurs à $e22$ dans l'ordre causal. Par analogie, l'ensemble des événements postérieurs à $e22$ dans l'ordre causal est $\{e23, e14\}$. Finalement, les événements $\{e12, e13, e23\}$, n'appartenant à aucun des deux ensembles précédents, sont des événements concurrents de $e22$.
- **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 différentes composantes agissent à des vitesses différentes. Cette différence de

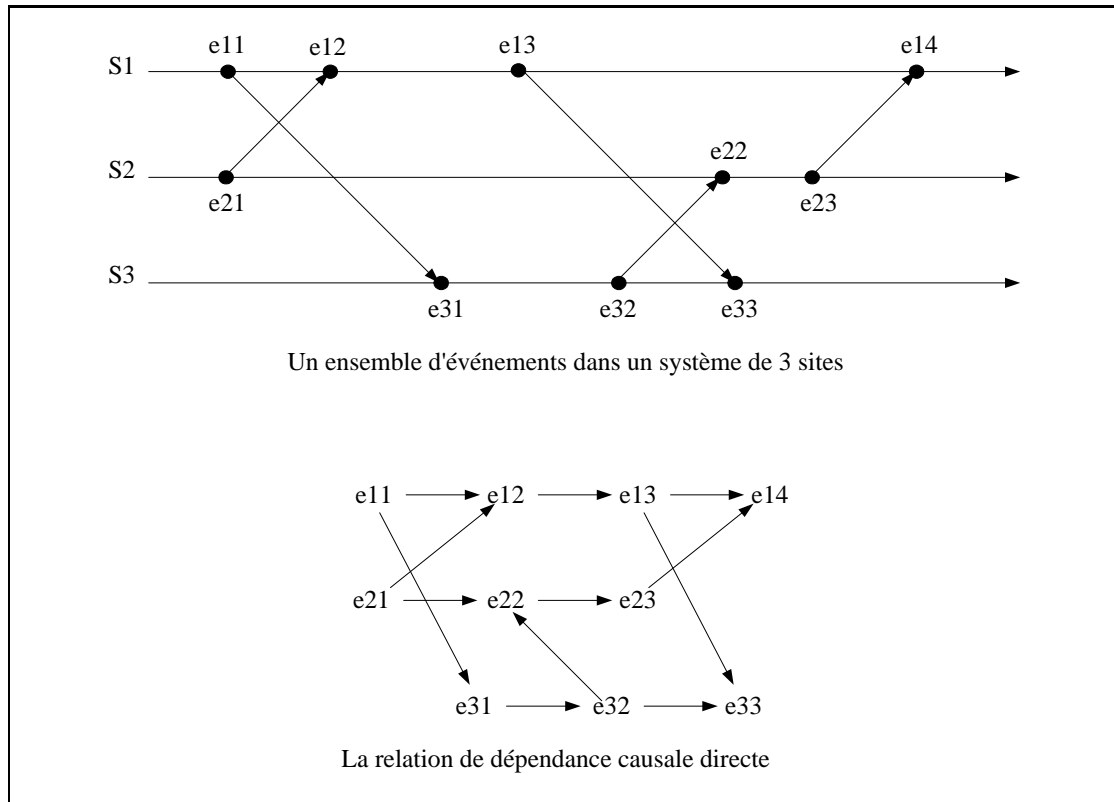


FIG. 1.2 – Horloge de Lamport

vitesse induit des comportements imprévisibles. A titre d'exemple, considérons trois sites A, B et C. Supposons que le site A possède une variable a de valeur 5 et que les deux sites B et C aient besoin d'effectuer une opération sur cette variable. Le site B envoie un message au site A pour obtenir la valeur courante de a . Le site C envoie lui un message au site A pour supprimer la variable a . Selon l'ordre de réception des messages, B obtiendra la réponse 5 (si son message arrive au site A avant celui de C) ou un message d'erreur (dans le cas contraire).

1.1.3 Problèmes classiques en algorithmique répartie

Le but des algorithmes répartis est de résoudre au mieux un problème réel. A travers la littérature, parmi les problèmes traités, on en trouve un petit nombre qui reviennent régulièrement. Dans cette section, nous dressons une liste non exhaustive de ces problèmes.

Les problèmes statiques

Les problèmes statiques sont ceux dont la spécification ne dépend que de l'état du graphe de communication du système. C'est généralement le cas lorsque les différents

processeurs présents sur le réseau collaborent au calcul de la taille du réseau, à l'élection d'un chef ou à la diffusion d'informations. La particularité de ces problèmes est qu'à partir d'un certain point de l'exécution, l'algorithme n'évolue plus.

Consensus. Le problème du consensus est un problème de base auquel se ramènent la plupart des problèmes d'accord (comme par exemple la diffusion atomique de messages ou la coordination d'actions réparties). L'objectif du consensus est de permettre à un ensemble de processus, chacun possédant sa valeur initiale, de décider de manière irrévocable sur l'une des valeurs initiales (par exemple pour identifier un processus en panne). Plus formellement, le consensus est spécifié par les propriétés suivantes :

- agrément : deux processus ne peuvent pas décider différemment ;
- terminaison : tout processus doit finalement décider ;
- validité : la valeur décidée est une des valeurs proposées ;

Une manière simpliste de résoudre le problème de consensus est que chaque processeur communique sa proposition ainsi que celles d'autres processus du groupe dont il a connaissance. Chaque processeur accumule les propositions des autres et se soumet à la proposition majoritaire.

Élection. Dans certaines applications réparties, il arrive qu'on ait besoin de distinguer un processeur des autres, appelé *leader*, pour servir de référence ou pour prendre des décisions globales concernant l'ensemble du réseau. Ainsi, le problème de l'élection consiste à élire, parmi tous les processeurs du système, un et un seul processeur dont tous les autres ont connaissance.

Diffusion d'information. Les systèmes répartis ne disposent *a priori* que d'un mode de communication point à point. Lorsqu'un processeur veut faire circuler une information, il ne peut le faire qu'en transmettant l'information aux processeurs ayant un lien de communication avec lui. Il s'agit donc de permettre à un processeur du système d'envoyer un message à un ensemble d'autres processeurs identifié comme un groupe. Pour cela, le problème consiste à élaborer des primitives de communication pour faire de *la diffusion* et de *l'échange total*. La diffusion consiste pour un processeur à faire parvenir son information à tous les autres. L'échange total est une généralisation de la diffusion, tous les processeurs communiquent leurs informations à tous les autres processeurs du système. Ces primitives rendent possible l'exploitation d'algorithmes, prévus initialement pour des systèmes de communication globale, dans des systèmes de communication point-à-point.

Les problèmes dynamiques

A l'inverse, les problèmes dynamiques ne dépendent pas uniquement de l'état du réseau mais aussi de son évolution dans le temps. En effet, les spécifications de ces problèmes portent sur les exécutions du système. C'est notamment le cas de l'exclusion mutuelle entre sections de code exécutées par les différents processeurs du réseau, ou encore la synchronisation des différentes copies d'une même information détenue par tous les processeurs. La particularité de ces problèmes est que toutes les exécutions du système sont infinies.

Exclusion Mutuelle. Pour éviter l'interférence des accès des différents processeurs à une ressource partagée, il faut assurer un accès exclusif à chaque fois. Ce mécanisme, appelé *exclusion mutuelle*, garantit un accès sérialisé et équitable à la ressource :

- *sûreté* : à chaque instant, un processeur au plus accède à la ressource ;
- *vivacité* : le temps d'attente d'un processeur pour accéder à la ressource est fini.

Ce genre de problème se pose notamment lors de la mise en commun de moyens de production, comme le partage d'une imprimante par plusieurs ordinateurs. Deux ordinateurs ne doivent pas pouvoir imprimer simultanément et tous doivent pouvoir accéder à l'imprimante ultérieurement. La méthode la plus simple pour résoudre ce problème est l'utilisation d'un jeton, qu'un processeur doit posséder pour accéder à la ressource, et qui va circuler dans le système.

Synchronisation. Dans les réseaux hétérogènes, les processeurs ne traitent pas l'information à la même vitesse. De même, les liens de communication ne font pas tous transiter l'information à la même vitesse. Ainsi, la nature des différents composants du système induit des écarts de vitesse entre ceux-ci :

- si on connaît des bornes maximales sur les délais de transmission de messages et les vitesses respectives d'exécution des processeurs les uns par rapport aux autres, le système est dit *synchrone* ;
- a contrario, si certains processeurs peuvent calculer plus vite que d'autres et que les délais de transmission de message sont finis mais pas bornés, le système est dit *asynchrone* ;
- entre ces deux extrêmes, un système est dit *partiellement synchrone* lorsqu'il existe des bornes mais qu'elles ne sont pas connues *a priori* ou effectives uniquement après un certain temps inconnu.

La synchronisation consiste à faire fonctionner, dans un système asynchrone, un algorithme réparti conçu pour un système synchrone. Pour cela, il faut utiliser un algorithme parallèle, appelé *synchroniseur*, simulant des processeurs synchrones dans un système asynchrone.

1.2 Hypothèses de modélisation

Suite à cette introduction portant principalement sur les enjeux et les caractéristiques des systèmes répartis, nous allons revenir au modèle, en décrivant plus précisément comment fonctionnent de tels systèmes. En effet, de par leur grande diversité, les systèmes répartis présentent des comportements très variables aussi bien du point de vue du mode de communication que du fonctionnement interne des processeurs. Néanmoins, il est possible de les classer en groupes selon certaines caractéristiques communes aux processeurs ou aux modes de communication.

1.2.1 Modes de communication

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 ni sur les capacités de chaque processeur à communiquer avec chacun des autres processeurs, ni sur la façon de communiquer. Bien qu'elles puissent se simuler l'une l'autre sous certaines hypothèses, on distingue généralement la communication *par échange de messages* de la communication *par mémoire partagée*.

Communication par échange de messages. Dans ce modèle, on considère qu'il existe des liens de communication entre les différents *nœuds*, par lesquels passent les informations. Lorsqu'un processeur veut transmettre une information à un autre processeur, il envoie un message via le lien de communication les reliant. L'hypothèse la plus restrictive est celle de la communication *globale*, où tous les processeurs sont connectés à un médium qui leur permet de communiquer entre eux. L'hypothèse la plus laxiste est celle de la communication *point-à-point*, où un lien permet à exactement deux processeurs de s'échanger des informations. De manière encore plus restrictive, un lien de communication ne permettant les échanges d'informations que dans un seul sens est dit *unidirectionnel*. A contrario, un lien *bidirectionnel* est un lien qui permet la communication dans les deux sens.

Par ailleurs, ce mode de communication prend en compte le temps qui s'écoule entre l'expédition d'un message et sa réception (délai de transmission) ainsi que les possibilités de perte ou d'altération des messages. L'échange de message est dit *synchrone* lorsque la communication se fait en un temps borné. Tout envoi de message se synchronise avec la réception correspondante pour ne plus former qu'une seule transition du système. Lorsqu'aucune garantie n'est donnée sur le délai d'acheminement des messages, on parle de communication *asynchrone*. Les messages sont alors stockés dans le lien jusqu'à ce que leur destinataire soit prêt à les recevoir.

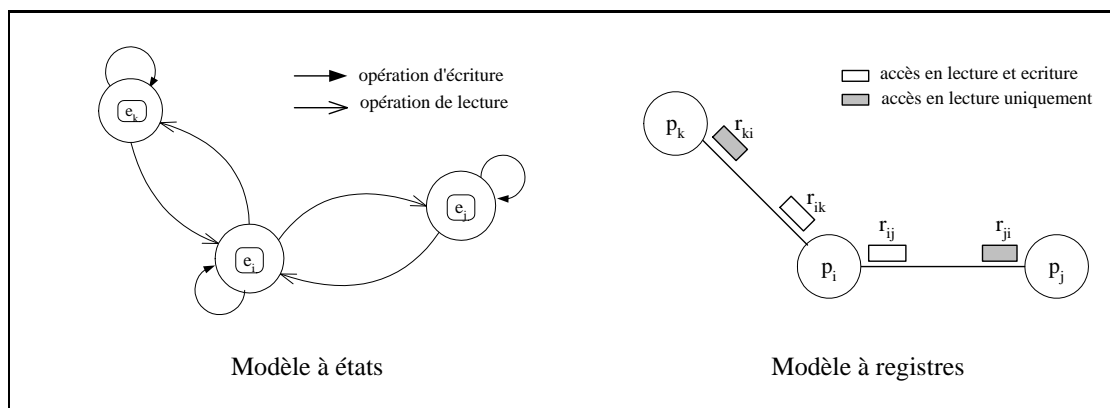


FIG. 1.3 – Modèles de communication par mémoires partagées

Communication par mémoire partagée. Dans le modèle de mémoire partagée, les processeurs disposent d'une zone mémoire commune dans laquelle ils peuvent lire (pour récupérer des informations) et écrire (pour passer des informations). Quand plusieurs processeurs partagent une même zone mémoire, certains peuvent se voir imposer des restrictions quant aux opérations qu'ils sont habilités à effectuer. Par exemple, un processeur peut disposer d'une variable ne pouvant être modifiée que par lui, mais partager toutefois cette variable en lecture avec un ou plusieurs autres processeurs. Pour ce qui est de la lecture des variables partagées, on peut considérer deux modèles différents : le modèle *read one* dans lequel un processeur ne lit que la variable d'un de ses voisins en une étape, et le modèle *read all* dans lequel un processeur lit les variables partagées de tous ses voisins en une seule étape. Par ailleurs, la figure 1.3 présente deux variantes de ce mode de communication :

- *le modèle à états*, dans lequel chaque processeur a un *état local* constitué de ses variables et de son compteur ordinal. Dans ce modèle, un processeur met à jour son propre état et accède directement aux états de ses voisins par des opérations de lecture.
- *le modèle à registres*, dans lequel la communication entre deux processeurs voisins se fait via deux registres (un registre chacun) partagés exclusivement entre ces deux processeurs. Par conséquent, chaque processeur dispose d'autant de registres que de voisins. Un processeur p_i peut lire et écrire dans ses propres registres r_{ij} , mais il ne peut que lire les registres de ses voisins. Ainsi, seuls les processeurs p_i et p_j ont le droit d'accéder aux registres r_{ij} et r_{ji} .

Les problèmes dus aux envois et réceptions de message ainsi que les difficultés rencontrées dans les preuves et les mesures de performance font que le modèle de mémoire partagée est souvent adopté dans la conception des algorithmes répartis. Remarquons

que la transformation de ce modèle en un modèle d'échange de messages est possible. En effet, la mémoire partagée peut être considérée comme un lien acceptant un seul message à la fois. A ce titre, de nombreuses études proposent des méthodes pour adapter à un autre modèle des algorithmes conçus pour un modèle particulier. Dolev, Israeli et Moran (1997a) proposent par exemple de simuler des algorithmes écrits en mémoire partagée à travers des algorithmes par échange de messages.

1.2.2 Topologies classiques

La topologie d'un système réparti peut être présentée par un graphe $G = (V, E)$ où l'ensemble V des sommets de G représentent l'ensemble des processeurs du système, et les arêtes E représentent les liens de communication entre les processeurs. Le choix d'une topologie dépend du problème traité et de l'avantage que représentent ses caractéristiques. La figure 1.4 présentent certaines topologies utilisés couramment dans les systèmes répartis du fait de leurs caractéristiques structurelles particulièrement avantageuses :

Anneau : un anneau à n sommets est un graphe où il existe une numérotation des sommets de s_0 à s_{n-1} telle que les arêtes sont de type (s_i, s_{i+1}) ou (s_{i+1}, s_i) (les indices sont pris modulo n). Cette forme de réseau est souvent utilisée du fait de sa simplicité, pour des algorithmes d'exclusion mutuelle. Avec cette topologie, tout processeur communique avec au plus deux voisins.

Arbre : un arbre à n nœuds est un graphe connexe à $n - 1$ arêtes. Cette définition implique qu'un arbre ne contient pas de cycle. Lorsque les liens sont bidirectionnels, la structure d'arbre assure l'existence d'un unique chemin entre tout couple de sommets du graphe. Les topologies en arbre sont utilisées dans le cadre distribué car elles permettent de minimiser les coûts de communication, de par le fait qu'elles minimisent le nombre d'arêtes utilisées. De plus, certains problèmes peuvent être résolus plus facilement sur un arbre, en utilisant par exemple des algorithmes par vagues qui permettent entre autre la diffusion d'information avec retour.

Étoile : un réseau est en étoile s'il possède un *nœud* central, et $n - 1$ arêtes connectant les $n - 1$ autres *nœuds* au centre. Cette topologie permet de représenter des réseaux où un processus joue le rôle de serveur pour tous les autres. Le problème des réseaux en étoile est que toutes les communications passent par le *nœud* central, au risque de ralentir le réseau, voir de la paralyser en cas de sa défaillance.

Certains algorithmes répartis sont conçus pour fonctionner avec des topologies bien particulières. Un problème se pose alors lorsque les systèmes réels sur lesquels sont implantés les algorithmes répartis ne possèdent pas la topologie requise. Il est dans ce cas nécessaire de faire appel à un autre algorithme qui construit virtuellement la topologie

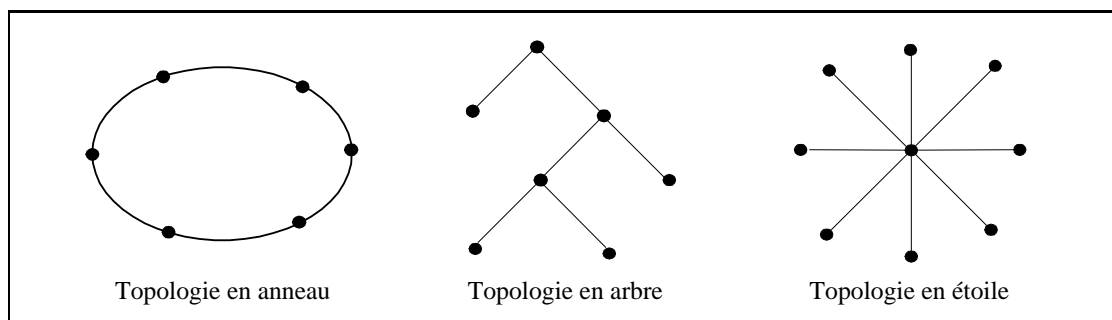


FIG. 1.4 – Quelques topologie classiques des réseaux

attendue par l'algorithme de plus haut niveau. Ainsi, de nombreux algorithmes répartis existent pour la construction de topologie particulières sur des graphes quelconques.

1.3 Tolérance aux défaillances

La sûreté de fonctionnement est un objectif inhérent aux systèmes répartis. En effet, l'augmentation du nombre des éléments constitutifs de tels systèmes contribue à l'augmentation de la probabilité que l'un de ces éléments tombe en panne. Cette multiplication des ressources réparties induit un besoin de tolérance aux pannes. De plus, les systèmes répartis fournissent souvent des services qui doivent rester accessibles en permanence, pendant toute la durée de vie du matériel. Dans les systèmes à grande échelle comme Internet, l'intervention d'un opérateur pour corriger les défaillances qui deviennent fréquentes est impossible, d'où la nécessité de mécanismes automatiques permettant aux systèmes de pallier à ces défaillances. Nous répertorions plusieurs types de défaillances suivant leur localisation (processeur ou lien), leur nature (intentionnelle ou accidentelle) et leur durée (permanente ou temporaire). Les techniques mises en oeuvre pour les tolérer sont alors différentes. Avant de présenter les diverses techniques, nous proposons une classification des différents types de défaillances qu'un système peut subir :

- Les *défaillances définitives* : elles sont souvent considérées comme les plus simples. Elles conduisent à l'arrêt du composant ou à la perte de son état interne. Dans le cas d'un processeur, cela signifie qu'il cesse de fonctionner de manière brutale et définitive. Dans le cas d'un lien, cela peut se traduire par une coupure définitive de la liaison de communication.
- Les *défaillances transitoires* : elles sont supposées apparaître de manière très espacées dans le temps, en alternance avec des périodes de temps suffisamment longues pendant lesquelles le système n'est pas sujet aux défaillances. Elles conduisent le composant à ne pas répondre à une sollicitation suite à une panne, puis à reprendre une activité normale immédiatement après la fin de cette panne. Dans le cas d'un

processeur, de telles défaillances se produisent par exemple lors d'une coupure temporaire de l'alimentation électrique. Dans le cas d'un lien de communication, des débranchements et branchements successifs lors d'une manipulation conduisent à des problèmes similaires.

- Les *défaillances byzantines* : elles sont totalement incontrôlables et demeurent les plus complexes à détecter et à résoudre. Elles conduisent à un comportement aléatoire du composant affecté. Un processeur a un comportement byzantin s'il ne suit pas les instructions de son algorithme. Les liens byzantins modifient les messages en transit par duplication, perte ou déséquence. Ce type de panne est le plus souvent lié à une erreur de programmation ou à une défaillance matérielle chronique.

Deux approches complémentaires se sont développées pour faire face aux différents types de pannes, l'une visant à garantir un fonctionnement correct des éléments non fautif du système, l'autre à rétablir un comportement correct après une panne. La première approche est appelée *tolérance aux pannes*. Lorsque le retour à la normale se fait sans l'aide d'un opérateur extérieur, la deuxième approche est appelée *auto-stabilisation*.

1.3.1 La tolérance aux pannes

L'approche classique pour faire face aux défaillances dans les systèmes répartis est appelée tolérance aux pannes. C'est une vision *pessimiste* qui consiste à suspecter toute information reçue et à vérifier constamment le comportement du système. Cette approche permet de masquer la présence de défaillances : celles-ci n'influencent pas le comportement du système. Elle est très efficace dans le cadre des défaillances définitives ou byzantines. Cependant, cette approche présente deux inconvénients :

- Afin de masquer la présence des défaillances, il est nécessaire de rajouter un contrôle, lors de chaque opération de communication, afin d'empêcher les défaillances de se propager et d'influencer la totalité du système. Malheureusement, ceci est relativement coûteux en terme de performances.
- Bien souvent, pour que les algorithmes tolérants aux fautes puissent fonctionner, il faut que certaines contraintes sur le nombre de composants atteints par des défaillances soient satisfaites : Lamport, Shostak et Pease (1982) ont établi que, même avec des systèmes de communication très fiables, il faudrait que moins d'un tiers des processeurs aient un comportement byzantin pour atteindre un agrément. En outre, Fischer, Lynch et Paterson (1985) ont montré qu'en cas de défaillance définitive touchant un seul processeur, il est impossible de résoudre le problème du consensus de manière déterministe dans un système asynchrone.

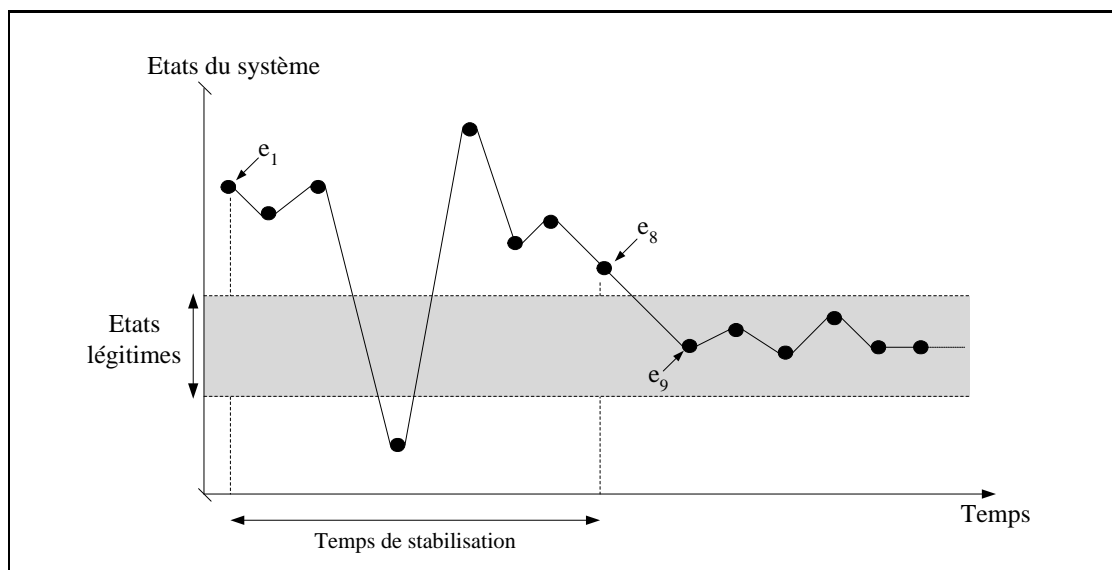


FIG. 1.5 – Une illustration des phases de l'auto-stabilisation

1.3.2 L'auto-stabilisation

A l'inverse, l'auto-stabilisation est une approche *optimiste* qui pallie aux défaillances transitoires. Elle a été introduite en 1974 par Dijkstra, mais sa portée dans le domaine de la résistance aux pannes n'a été soulignée que bien plus tard par Lamport (1984). Cette approche tolère une interruption temporaire du service du système et un comportement inconsistant de certains processeurs, mais garantit un retour à la normale au bout d'un temps fini. Ainsi, après un ensemble de corruptions, le système est dans un état quelconque. La propriété d'auto-stabilisation garantit que le système atteindra au bout d'un temps fini une configuration à partir de laquelle il se comportera suivant sa spécification.

Le comportement d'un système auto-stabilisant est le suivant : suite à une panne, le système traverse une phase transitoire, appelée *phase de stabilisation*, au cours de laquelle il peut ne pas être conforme à sa spécification (on parle alors d'états illégitimes du système, ou d'états incorrects). Après un temps fini et en l'absence de nouvelles pannes, le système est dans une *phase stabilisée* dans laquelle il est de nouveau conforme à sa spécification (on parle alors d'états légitimes du système, ou d'états corrects). Une illustration de ces différents aspects est donnée sur la figure 1.5. Le système part d'un état quelconque, en l'occurrence l'état illégitime e_1 . Durant la phase de stabilisation, le système oscille entre un certain nombre d'états illégitimes, en l'occurrence les états e_1 à e_8 , puis retrouve un comportement correct en atteignant la configuration légitime e_9 . A partir de cet instant et en l'absence de nouvelle panne, le système restera dans l'ensemble des états légitimes (phase stabilisée). Il faut préciser que le retour à un comportement normal se fait sans

aucune aide extérieure, que ce soit pour détecter la panne ou pour y remédier.

Exemple 1.1 Exclusion mutuelle sur un anneau. *Considérons le système réparti composé de 6 machines organisées en un anneau unidirectionnel présenté sur la figure 1.6. Chaque machine dispose d'un compteur à valeur dans $[0 \dots n - 1]$ et peut lire la valeur du compteur de la machine précédente sur l'anneau. Les différentes machines se partagent une imprimante et veulent y accéder simultanément. Ce problème est connu sous le nom d'exclusion mutuelle : plusieurs processeurs (les machines) veulent accéder de manière exclusive à une même ressource (l'imprimante). Une première solution consisterait à faire circuler un jeton¹ sur le réseau. La machine possédant le jeton peut imprimer.*

Si une panne vient perturber ce système en engendrant des jetons surnuméraires, un algorithme qui n'est pas auto-stabilisant conduira le système à une situation où la propriété de sûreté n'est pas assurée. En effet, la présence d'un jeton sur une machine lui donnant le droit d'accéder à l'imprimante, la présence de plusieurs jetons implique un accès simultané et concurrent de plusieurs machines. C'est la raison pour laquelle, si plusieurs jetons sont présents dans un état du système, il est nécessaire que toute exécution conduise invariablement le système vers un état où seul un jeton subsiste. Une solution auto-stabilisante proposée par Dijkstra (1974) permet une disparition progressive des jetons surnuméraires. Elle consiste à donner à l'une des machines le rôle de "filtre à jeton surnuméraire". L'accès à l'imprimante est matérialisé par la possession du jeton qui est conditionnée comme suit :

- pour la machine distinguée D , la valeur de son compteur doit être égal à celle du compteur de la machine précédente sur l'anneau.
- pour toute autre machine, la valeur de son compteur doit être différente de celle de la machine précédente sur l'anneau

Qu'elle soit intéressée ou pas par l'imprimante, l'action d'une machine consiste à transmettre le jeton à la machine suivante sur l'anneau. De plus, seule la machine D peut mettre à jour la valeur du jeton avant de le transmettre. En effet, la valeur d'un jeton transmis par la machine D doit indiquer la valeur courante du compteur de la machine D pour que cette valeur soit transmise à toutes les autres machines du réseau. Pour cela, chaque machine modifie la valeur de son compteur de manière à ce qu'elle n'ait plus de jeton, de la façon suivante :

- la machine distinguée D incrémente la valeur de son compteur modulo n .
- toute autre machine recopie la valeur du compteur de la machine précédente sur l'anneau.

Une exécution possible de cette solution auto-stabilisante est décrite sur la figure 1.6, où le nœud en gras représente la machine distinguée D et les nœuds grisés celles qui possèdent un jeton. Dans un état correct du système, toutes les machines ont leurs compteurs évalués à 4 et une seule machine détient l'unique jeton \bullet : supposons que ce soit la machine D . Bien évidemment, la valeur contenue dans ce jeton est 4. A l'étape 0, supposons qu'une panne transitoire frappe les trois machines M_2 , M_3 et M_4 du réseau modifiant

¹Le jeton est une séquence de bits qui circule constamment d'un processeur à l'autre sur le réseau. Pour qu'un processeur puisse accéder à une ressource ou envoyer un message, il doit attendre le passage du jeton.

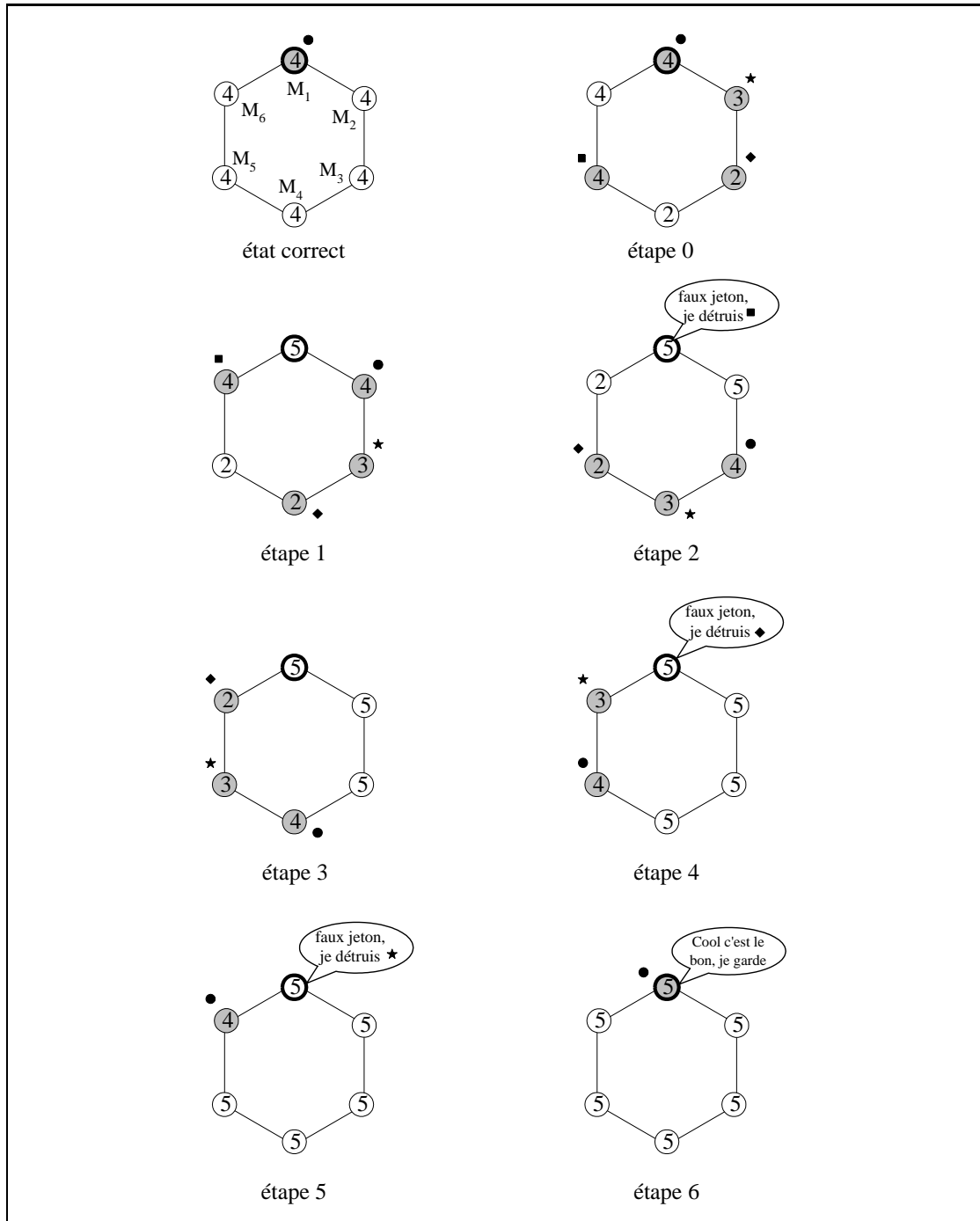


FIG. 1.6 – Phase de stabilisation de l'algorithme d'exclusion mutuelle de Dijkstra

ainsi la valeur du compteur de M_2 à 3 et celles des compteurs de M_3 et M_4 à 2. Ce faisant et suivant les conditions de possession d'un jeton, cette panne engendre 3 jetons supplémentaires : un sur la machine M_2 (marqué \star et évalué à 3), un deuxième sur la machine M_3 (marqué \blacklozenge évalué à 2) et un troisième sur la machine M_5 (marqué un troisième jeton \blacksquare évalué à 4). Au cours de la phase de stabilisation et suivant les instructions de l'algorithme, la machine distinguée D incrémente son compteur à l'étape 1 et met à jour la valeur du jeton \bullet (qui devient égale à 5) avant de le transmettre à la machine suivante. De même, chacune des machines détenant un jeton met à jour la valeur de son compteur et passe le jeton à la machine suivante sur l'anneau.

A l'étape 2, la machine D voit passer le jeton \blacksquare avec une valeur 4 différente de celle de son compteur. Dans ce cas, le jeton \blacksquare ne franchira pas la machine D qui conclura qu'il est un jeton surnuméraire et le supprimera. Ainsi à l'étape 3, il ne reste que 3 jetons circulant entre les machines.

A l'étape 4, le jeton \blacklozenge évalué à 2 connaîtra le même sort que le jeton \blacksquare , il sera supprimé par D . De même pour le jeton \star évalué à 3, il sera supprimé à l'étape suivante. Le jeton restant se trouve sur la machine M_6 dont la valeur du compteur est 4.

A l'étape 6, le système est à nouveau dans un état correct : M_6 met son compteur à 6 et transmet le jeton \bullet évalué à 5 à la machine suivante sur l'anneau, la machine D . La valeur du jeton étant la même que celle du compteur de D , cette dernière conservera le jeton pour un nouveau tour.

A la différence de l'approche classique, les défaillances ne sont pas masquées : pendant la phase de stabilisation, le système ne vérifie pas sa spécification. D'autres différences résident dans le type de défaillances supportées et les hypothèses nécessaires pour garantir la tolérance aux défaillances. L'auto-stabilisation tolère les pannes sous couvert de deux hypothèses :

- un code fiable : les pannes supportées (les corruptions mémoires) sont supposées n'affecter que la partie volatile des composants du système, à savoir la mémoire vive des processeurs et les messages en transit dans les liens de communications. Le code du programme et les informations contenues dans la mémoire permanente, quant à eux, ne sont pas affectés.
- une faible fréquence des pannes : les pannes frappant le système doivent être suffisamment éloignées dans le temps les unes des autres pour que le système puisse retrouver un comportement correct entre deux pannes. En effet, si les pannes sont fréquentes, le système passe à la phase de stabilisation et y demeure, vu que l'intervalle de temps séparant deux pannes consécutives est insuffisant pour conduire le système vers un état correct.

Avantages de l'auto-stabilisation

Outre la tolérance aux pannes transitoires, un des avantages majeurs de l'auto-stabilisation réside dans l'absence d'initialisation. Les algorithmes auto-stabilisants sont conçus pour fonctionner comme s'il n'y avait jamais de défaillance, mais pour garantir un comportement correct même avec une mauvaise initialisation. En effet, aucune restriction n'étant faite sur le nombre de processus sujets aux défaillances, tous les processus peuvent être simultanément sujets aux défaillances. Dans ce cas, le système est totalement corrompu et son état initial est considéré comme étant quelconque. Puisque la convergence est garantie depuis toute configuration initiale, même si chaque processeur démarre son algorithme dans un état quelconque, le système est capable de retrouver un comportement correct.

Un autre avantage des algorithmes auto-stabilisants est qu'ils fonctionnent pour des topologies de réseaux dynamiques. En effet, les corruptions transitoires supportées peuvent être des changements de topologie (e.g. ajout de nouveau processeur dans le système), des suites d'arrêts avec redémarrage ou des comportements malicieux pendant un temps fini. Suite à ce type de défaillances, le système peut atteindre un état qui n'est pas un état correct. En l'absence de nouveau changement dans la topologie pendant un temps suffisamment long pour que la convergence soit atteinte, le système se comportera de manière correcte au bout d'un temps fini.

Limites de l'auto-stabilisation

Malgré leurs avantages, les algorithmes auto-stabilisants présentent plusieurs inconvénients :

- un premier inconvénient réside dans la définition de l'auto-stabilisation : quel que soit l'état initial, le système se comporte correctement au bout d'un temps fini, mais aucune propriété n'est assurée durant la phase de stabilisation. Cette absence de contraintes oblige le système à être capable de supporter la non conformité à ses spécifications pendant un certain temps.

Exemple 1.2 *Nous illustrons cette problématique en prenant l'exemple d'un réseau étendu, qui est un ensemble de réseaux locaux, reliés entre eux par des routeurs. La communication d'un réseau local à un autre se fait point-à-point, par envoi des messages à un premier routeur, qui les transmet lui-même à un second, et ainsi de suite jusqu'à ce que les messages atteignent un routeur qui peut communiquer directement avec la machine destinataire. Le travail d'un routeur est donc de chercher des chemins entre différents points d'un réseau plus ou moins complexe. Ces chemins vont notamment servir à la circulation d'informations. Un protocole de routage construit les tables de routage : ce sont des tables de correspondance entre l'adresse de la machine destinataire et le routeur suivant dans*

l'acheminement du message. Ces protocoles cherchent toujours le chemin le plus efficace possible entre deux points déterminés, en tenant compte de tout ce qui pourrait se produire dans la vie courante des réseaux. Dans des réseaux simples, si un ordinateur tombe en panne, alors le seul moyen de rediriger les paquets vers la bonne destination est une intervention manuelle. Dans des réseaux complexes comme Internet, une telle intervention n'est pas possible car les pannes ont lieu de façon imprévisible (i.e., les pelleteuses éprouvant une certaine attirance pour les câbles), et concernent souvent un grand nombre de routeurs (i.e., les fournisseurs de service rajoutant des nouvelles fibres optiques pour améliorer le trafic). Il serait alors fastidieux de trouver la panne et de rediriger manuellement les paquets à acheminer. Pour cela, il existe des techniques diverses de mise à jour automatiques des tables de routage : ce sont les protocoles de routage dynamique tels que RIP et OSPF. La nouvelle situation, dans laquelle la topologie du réseau a changé et les tables de routage ne sont plus exactes peut être considérée comme un état initial du protocole de routage. Si les défaillances ne se produisent plus pendant suffisamment longtemps, la propriété d'auto-stabilisation garantit que les tables de routage finiront par être corrigées. Par contre pendant ce temps, les tables de routage peuvent être fausses et donc des messages peuvent être perdus ou envoyés vers d'autres destinations.

- un deuxième inconvénient réside dans l'incapacité de détecter la terminaison de l'algorithme. Comme chaque processus n'a qu'une vision locale de l'état du système, les processus ne sont pas capables de s'apercevoir qu'une configuration légitime a été atteinte. Ainsi, ils ne sont jamais avertis du fait que leur fonctionnement est devenu fiable. Cela les oblige à exécuter leur code en permanence. Ceci implique, en l'absence de pannes transitoires, un surcoût de fonctionnement des algorithmes auto-stabilisants par rapport à leurs équivalents non stabilisants.

- un troisième inconvénient est l'absence de garantie sur le temps de stabilisation. En effet, l'auto-stabilisation peut être comparée à un service après-vente. Le concept est intéressant si les délais de réparation sont raisonnables. Pour qu'un système auto-stabilisant fournisse son service le plus rapidement possible, il faut limiter son temps de stabilisation. Plus ce temps est court, plus le service est rendu effectif rapidement. De plus, un court temps de stabilisation réduit la probabilité d'occurrences d'autres défaillances avant le rétablissement du service (ce qui empêcherait le système de se stabiliser). Or, l'auto-stabilisation n'impose aucune contrainte sur le temps de convergence : la seule garantie est que le système atteindra, au bout d'un certain temps de stabilisation, une configuration à partir de laquelle le service sera rendu. Ceci est vrai quelle que soit la gravité de la défaillance : elle peut affecter une partie ou la totalité des composants du système. Par ailleurs, les algorithmes auto-stabilisants utilisent les mêmes mécanismes de correction en cas de panne légère qu'en cas de panne totale. Même si un seul composant est sujet à une défaillance, la valeur corrompue peut se propager et corrompre tout le reste du réseau, avant qu'il n'atteigne de nouveau un comportement correct.

Exemple 1.3 *Pour illustrer cette problématique, considérons le problème de construction des tables de routage au-dessus d'un graphe de communication. Nous nous restreignons à une destination particulière appelée racine et à un pas intermédiaire, qui consiste à pondérer chaque noeud par sa distance à la racine. L'objectif est de construire un arbre de plus courts chemins vers la racine. Le principe de l'algorithme est extrêmement simple : le processeur destination (la racine de l'arbre) diffuse une distance 0 à tous ses voisins ; tout autre processeur calcule sa distance à la racine en fonction des distances supposées de ses voisins, et la diffuse lui-même à ses voisins. Cet algorithme est auto-stabilisant mais la moindre défaillance, même si elle n'affecte qu'un petit nombre de routeurs, peut mener avant stabilisation à la perturbation d'une grande partie du système. Chaque état du système correspond à une affectation de valeurs aux différents noeuds du réseau (où chaque routeur est un nœud et chaque lien de communication un lien entre deux routeurs). L'état de chaque routeur est représenté par un entier, qui exprime sa distance à la racine (valuée par 0). La première configuration représente un état correct du système, où tous les routeurs qui connaissent leur distance exacte et correcte de la racine.*

A l'étape 0, considérons qu'une panne transitoire frappe l'un des routeurs du système de distance 5 à la racine, et indique qu'il est à distance 0 de tous les autres routeurs du réseau sur la figure 1.7. Chaque routeur communique à ses voisins les modifications intervenues dans sa table de routage dès qu'elles sont connues. Dans le cas d'un réseau asynchrone ou sous l'hypothèse que le routeur défaillant soit surchargé, il ne reçoit pas l'information des autres routeurs pendant un certain moment. Entretemps, les routeurs voisins du routeur fautif reçoivent son information.

Lors des étapes 1 et 2, les autres routeurs mettent à jour leurs distances en considérant le routeur défaillant comme le meilleur passage pour toute destination et de plus en plus de messages vont être dirigés vers lui. Ainsi, la défaillance se propage dans le système, simplement en respectant les règles de l'algorithme. Au bout d'un certain temps, ce routeur défaillant finira par avoir une table de routage correcte, mais cette unique défaillance aura perturbé une part non négligeable du système. Ainsi, la défaillance n'a pas été masquée et elle a influencé de manière significative le système. Intuitivement, on comprend que tous les routeurs devant passer par le routeur défaillant, pour atteindre la racine, soient influencés par cette défaillance, mais on constate que d'autres routeurs autour le sont également.

A partir de l'étape 3 et sous l'hypothèse que le routeur fautif est moins surchargé, l'auto-stabilisation du système débute. Précisons qu'on suppose que tous les routeurs (y compris le fautif) travaillent au même rythme et que les messages transitent à la même vitesse. A l'étape 8, tous les routeurs ont corrigé leurs distances.

Pour palier à cette dernière limite, on serait en droit d'espérer un traitement plus local et plus rapide des fautes singulières (c'est-à-dire qui n'affecte qu'un seul processeur), permettant de les circonscrire à une zone proche des processeurs corrompus. C'est l'objet de la prochaine section.

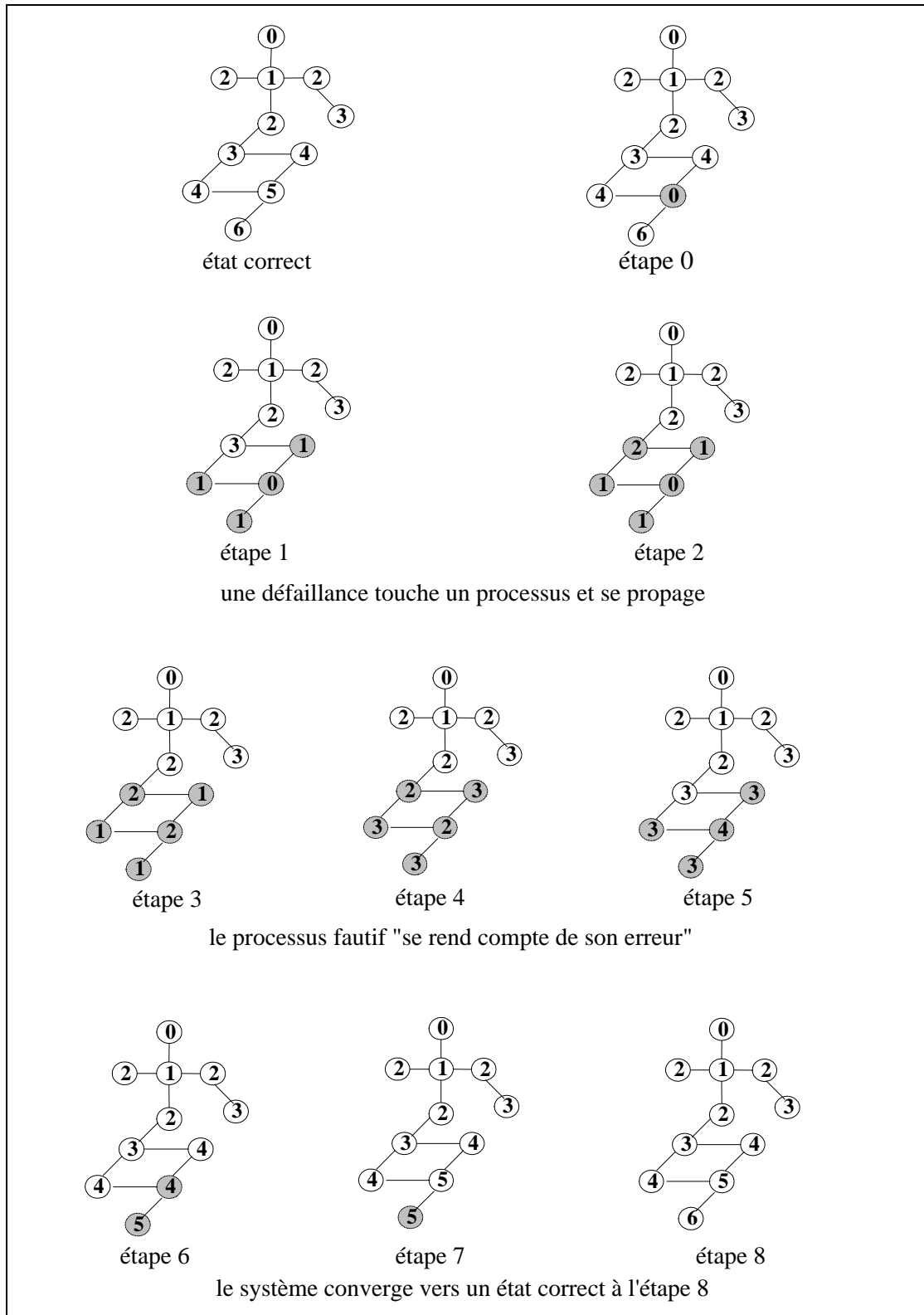


FIG. 1.7 – Une illustration de la propagation d'une seule faute

1.3.3 L'auto-stabilisation améliorée

Pour contenir les défaillances et contraindre le temps de stabilisation en fonction de leurs gravités, il est intéressant de rajouter des mécanismes sur les protocoles auto-stabilisants pour corriger les défaillances transitoires quand celles-ci sont peu nombreuses. Nous présentons maintenant trois approches qui se sont développées dans cette optique.

La k -stabilisation

La k -stabilisation est une généralisation de l'auto-stabilisation. Elle a été définie et présentée pour la première fois dans Beauquier, Genolini et Kutten (1998). L'idée de base est d'obtenir des temps de convergence courts, en faisant l'hypothèse qu'une petite partie du réseau seulement est corrompue. Les protocoles k -stabilisants sont des protocoles stabilisants pour le cas où une borne supérieure $k \leq n$ sur le nombre de fautes est connue, n étant le nombre total de processeurs dans le système. Plus précisément, sur un réseau de taille n , un algorithme k -stabilisant tolère jusqu'à k fautes. Ainsi, un algorithme k -stabilisant est conçu pour résister à des pannes affectant au plus k processeurs. Dans le cas où k est la taille du réseau, les concepts de k -stabilisation et d'auto-stabilisation sont identiques. Dans le cas inverse, un système k -stabilisant frappé par une panne affectant plus de k processeurs n'est pas sûr de se stabiliser. L'intérêt de ces algorithmes est d'avoir, dans le cas où un faible nombre de fautes frappent le réseau, un temps de convergence plus court que leurs versions auto-stabilisantes. A titre d'exemple, considérons le problème de l'exclusion mutuelle sur un anneau de taille n . Si k fautes frappent le réseau, ce problème admet une solution auto-stabilisante convergeant en $k \times n$ étapes, alors qu'un algorithme k -stabilisant (Beauquier et al., 1999a) résout le même problème en k^2 étapes.

Par ailleurs, les algorithmes k -stabilisants ne présentent aucune garantie sur le temps de stabilisation. En particulier, lorsqu'un petit nombre de fautes frappe le réseau, un algorithme k -stabilisant converge mais lentement. D'où l'émergence d'un nouveau concept : les *algorithmes adaptables en temps* (aussi appelés *algorithmes proportionnels*) pour remédier à cet inconvénient. Le temps de convergence de ces algorithmes s'adapte au nombre de défaillances touchant le système.

L'adaptabilité en temps

L'idée de relier le temps de convergence au nombre de défaillances est appelée la stabilisation relative en temps (*time adaptive*), ou proportionnelle (*fault local*). Elle a été introduite par Kutten et Patt-Shamir (1997) pour des problèmes non réactifs². Les

²Ce sont les problèmes dont la spécification consiste à obtenir une fonction de la topologie et des données en entrée, comme le problème de persistance de bit, de coloration ou d'agrément.

auteurs proposent un transformateur des algorithmes non réactifs garantissant aux algorithmes transformés un temps de stabilisation proportionnel au nombre de défaillances. Pour les problèmes dynamiques, Afek et Dolev (1997) ont proposé un transformateur basé sur des prises de vues progressives, puis Kutten et Peleg (1999) et Beauquier, Genolini et Kutten (1999a) ont proposé des algorithmes proportionnels pour le problème de circulation de jeton. La stabilisation proportionnelle assure un retour à la normale en un temps proportionnel au nombre effectif de fautes. Autrement dit, si le réseau est touché par f corruptions, son temps de convergence dépendra de f et non d'une constante k ou de la taille n du réseau. Cela signifie que la phase de stabilisation sera très rapide dans le cas d'une unique faute, tout en restant acceptable dans le cas de la corruption de toute une partie du réseau. Elle garantit également toutes les propriétés propres à l'auto-stabilisation. La phase de stabilisation sera peut-être longue (si une grande partie du réseau est touchée), mais le système convergera vers un fonctionnement correct, comme c'est le cas pour l'auto-stabilisation.

Par ailleurs, la taille de la zone de propagation des erreurs est bornée. En effet, les transmissions de messages entre processeurs prennent du temps. La phase de stabilisation étant courte, un processeur physiquement éloigné d'un lieu de défaillance n'aura donc pas le temps de recevoir un message en provenance d'un processeur corrompu. En cas de faible corruption, les erreurs ne peuvent se propager loin du lieu de défaillance.

Le fait que le processus de récupération sur erreur puisse impliquer le réseau tout entier empêche son utilisation dans des réseaux de grande taille tels qu'Internet. Pour permettre le passage à l'échelle, il faudrait que le temps de reprise sur erreur soit d'autant plus faible que le nombre de fautes touchant le réseau est réduit. Ghosh et He (2002) présentent, pour des problèmes non réactifs dans un système synchrone, un algorithme proportionnel qui se stabilise en $O(k^2)$ (où k est le nombre de fautes). Leur méthode consiste à décomposer le réseau en régions fautives, et ensuite à mesurer et à comparer ces régions fautives pour ordonnancer les actions du système de façon à diminuer progressivement la taille de ces régions jusqu'à leur disparition.

Le confinement de fautes

Même si la stabilisation proportionnelle limite la propagation des erreurs, elle ne peut totalement éviter la corruption d'une partie plus ou moins grande du réseau. Classiquement, les algorithmes proportionnels tolèrent des pannes pouvant affecter la totalité d'un système réparti. On serait en droit d'espérer un traitement plus local et plus rapide des fautes singulières, ainsi que leurs circonscription à une zone proche des processeurs corrompus. Un algorithme confinant les fautes est un algorithme qui a la capacité de limiter les effets des fautes transitoires à un petit nombre de composants du système, et

de converger le plus rapidement possible. Ceci permet de masquer les fautes vis-à-vis de l'utilisateur du système. A titre d'exemple, dans des réseaux de grande taille, tel qu'Internet, il est souhaitable qu'une faute transitoire frappant un seul routeur soit réparée le plus rapidement possible et n'affecte qu'un petit nombre de routeurs. Ainsi, la faute est transparente pour l'utilisateur du réseau. Autrement dit, nous désirons que la perturbation causée par une faute soit proportionnelle à sa gravité. Ceci n'étant pas toujours possible, Ghosh et Gupta ont introduit (Gupta, 1997; Ghosh et Gupta, 1996; Ghosh et al., 1996a,b, 1997) la notion de confinement de fautes (*fault-containment*), qui consiste à prendre des précautions particulières pour empêcher les défaillances de se propager dans le système. Ils ont également introduit une mesure de complexité, le nombre de contaminations, qui correspond au nombre de processeurs influencés par les défaillances.

Le confinement de fautes est une spécification de la stabilisation proportionnelle. Dans le cas d'une faute singulière (c'est-à-dire touchant un seul site du réseau), un algorithme de confinement de fautes converge en $O(1)$ vers un état correct. Il garantit aussi la stabilisation en cas d'une faute frappant plus d'un processeur. Une généralisation du confinement de fautes est le k -confinement de fautes. Un algorithme k -confinant les fautes converge en $O(f(k))$ avec $f(k)$ une fonction monotone croissante du nombre k de processeurs fautifs.

1.4 Positionnement de notre travail

Comme indiqué précédemment, les deux principales approches pour pallier aux différents types de défaillances dans un système réparti sont :

- La tolérance aux pannes : approche pessimiste qui supporte les pannes définitives. Elle consiste à garantir quoi qu'il arrive un fonctionnement correct des éléments non fautifs.
- L'auto-stabilisation : approche optimiste qui supporte les pannes transitoires. Elle consiste à garantir un fonctionnement correct la plupart du temps. Une interruption temporaire du service est acceptée, mais avec une garantie de retour à la normale au bout d'un temps fini.

Nous nous intéressons dans cette thèse à la deuxième approche. Plus précisément, nous allons essayer d'appliquer les principes de l'auto-stabilisation dans des réseaux de robots mobiles, alors que les travaux existant portent essentiellement sur des réseaux filaires.

Étant donné que l'auto-stabilisation permet de tolérer un nombre arbitrairement grand de défaillances, cette technique souffre de plusieurs limites, parmi lesquelles on peut mentionner le processus de récupération sur erreur, susceptible de perturber tout le réseau même si un petit nombre de processeurs seulement sont défaillants. D'où le

besoin de *raffinement* de l'auto-stabilisation et le développement de techniques telles que la k -stabilisation, l'adaptabilité relative en temps ou encore le confinement de fautes. Ces techniques consistent à concevoir des algorithmes basés sur les mêmes concepts de l'auto-stabilisation, mais offrant des temps de convergence plus courts que leurs équivalents auto-stabilisants.

Aussi, nous proposons une application originale de la technique de confinement de fautes. Plus précisément, nous développons un algorithme pour la résolution du problème de l'élection d'un leader dans un réseau avec identifiants. Même si la technique de confinement de fautes offre des temps de convergence proportionnels au nombre de fautes, et que le problème de l'élection est l'un des problèmes classiques en contrôle du réseau, ce champ d'investigation a été peu exploré jusqu'à aujourd'hui.

Chapitre 2

Modèle

Résumé. *Dans le cadre de l'étude des algorithmes répartis et en fonction des besoins d'expression et de formalisme, plusieurs modèles différents par bien des aspects sont proposés. Dans ce chapitre, nous formalisons certains concepts utiles pour présenter des algorithmes répartis et démontrer leurs propriétés. Dans un premier temps, nous rappelons quelques notions fondamentales de la théorie des graphes et des systèmes de transitions. Puis, nous décrivons un système réparti à travers ses composants de base, les processeurs et les liens de communication qui sont synchronisés entre eux. De plus, comme il est d'usage, nous introduisons les notions de configuration, de transition et de démon qui servent à décrire les exécutions d'un système réparti. Nous présentons formellement la notion d'auto-stabilisation. Nous précisons enfin les mesures de complexité associées aux algorithmes auto-stabilisants ainsi que les mesures de complexité plus particulières aux algorithmes confinant les fautes.*

2.1 Introduction

Les systèmes répartis ont été largement étudiés et modélisés sous forme de graphes dont les sommets et les arêtes sont respectivement les processeurs et les liens de communication. La modélisation de ces composants de base d'un système se fait soit par des systèmes de transitions comme dans Attiya et Welch (1998) et Tel (1994), soit par des automates d'entrées/sorties comme dans Lynch (1996). Dans le cadre de l'auto-stabilisation, d'autres modèles ont été développés lors de travaux antérieurs. Herman (1991) a proposé un système de transition à atomicité fine (cf. section 2.4.2) et Petit (1998) un modèle à états basé sur le système de transition développé dans Tel (1994). Par ailleurs, Tixeuil (2000), Grandinariu (2000) et Herault (2003) proposent des modèles basés sur des systèmes de transitions proche de celui décrit dans Attiya et Welch (1998). Pour plus d'informations sur les systèmes répartis et leur modélisation, le lecteur pourra se reporter à l'ouvrage de Dolev (2000) et à l'article de synthèse de Schneider (1993).

2.2 Notions préliminaires

Dans cette section, nous présentons les principales notations utilisées pour la description d'un système réparti. Dans un premier temps, nous rappelons quelques notions fondamentales de la théorie des graphes afin de définir les notions de topologie et de voisinage. Ensuite, nous décrivons les systèmes de transitions, qui servent en particulier à la modélisation de l'évolution des composants de base des systèmes répartis.

2.2.1 Graphes

Définition 2.1 (Graphe non orienté) *Un graphe non orienté G est un couple (V, E) où V est un ensemble non vide de nœuds, et $E \subseteq V \times V$ un ensemble de couples $\{u, v\}$ de nœuds distincts, appelées arêtes.*

L'ensemble E définit une relation de voisinage entre les éléments de l'ensemble V . Deux nœuds u et v de V sont dits voisins si $\{u, v\} \in E$.

Définition 2.2 (Graphe orienté) *Un graphe orienté G est un couple (V, E) où V est un ensemble non vide de nœuds, et $E \subseteq V \times V$ un ensemble de couples ordonnées (u, v) , appelées arcs. Si (u, v) est un élément de E , alors il existe un arc sortant de u et entrant dans v .*

Le degré *entrant* d'un nœud $u \in G$, dénoté $\Gamma^-(u)$, est le nombre de nœuds v tels que l'arc $(v, u) \in G$. De manière similaire, le degré *sortant* d'un nœud $u \in G$, dénoté $\Gamma^+(u)$, est le nombre de nœuds v tels que l'arc $(u, v) \in G$. Dans le cas d'un graphe G non orienté, le degré d'un nœud u de G , dénoté $\Gamma(u)$ est égal au nombre de nœud v tels que l'arête (u, v) est dans G . Le degré d'un graphe est le plus grand degré de ses nœuds.

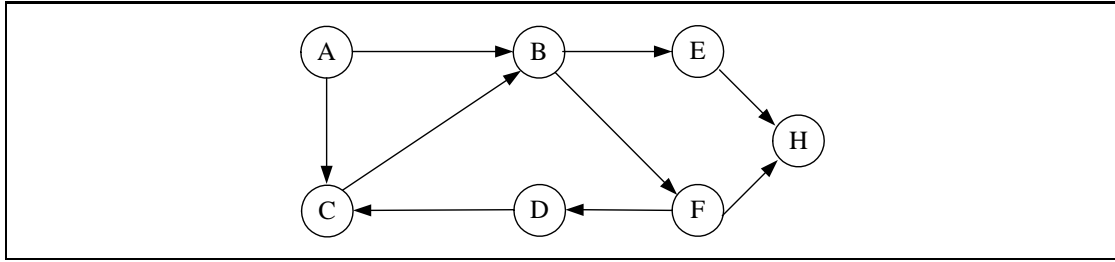


FIG. 2.1 – Un exemple de graphe orienté

Définition 2.3 (Voisin) Un nœud v est un voisin de u s'ils sont connectés par un arc ou une arête. Nous notons Voisins_u l'ensemble des voisins de u .

Le nombre des voisins d'un nœud u est noté $\Gamma^+(u) \cup \Gamma^-(u)$

Exemple 2.1 Considérons le graphe orienté G de la figure 2.1 comportant 7 nœuds et 9 arcs. L'ensemble des nœuds est $V = \{A, B, C, D, E, F, H\}$ et l'ensemble des arcs est $E = \{(A, B), (A, C), (B, E), (B, F), (C, B), (D, C), (E, H), (F, D), (F, H)\}$. Le degré sortant du nœud A est 2 car seuls les arcs (A, B) et (A, C) sont dans l'ensemble des arcs de G . Le degré entrant du nœud A est 0 car il n'existe aucun arc d'extrémité A dans l'ensemble des arcs de G . Les nœuds B, D et H sont les voisins du nœud F . Le degré du graphe G est 4 car B est le nœud de plus grand degré, 4.

Définition 2.4 (Chemin) Soit $G = (V, E)$ un graphe (orienté ou non), et u et v deux nœuds de V . Il existe un chemin entre u et v si et seulement si il existe $n_0, n_1, n_2, \dots, n_i$ éléments de V tels que :

- $\{u, n_0\} \in E, \{n_i, v\} \in E$ et,
- $\forall 0 \leq j < i, \{n_j, n_{j+1}\} \in E$.

$\{u, n_0, n_1, n_2, \dots, n_i, v\}$ est alors appelé chemin entre u et v .

Si chaque n_i est unique le long du chemin, le chemin est alors dit *élémentaire*. Un *cycle* est un chemin où $u = v$. Un graphe non orienté est appelé acyclique s'il ne contient aucun cycle de longueur 3 ou plus. Un graphe orienté sans cycle est appelé un DAG (de l'anglais, Directed Acyclic Graph).

Définition 2.5 (Longueur) La longueur du chemin $\{u, n_0, n_1, n_2, \dots, n_i, v\}$ est le nombre d'arêtes ou d'arcs qu'il faut franchir pour aller de u à v le long de ce chemin, soit $i + 2$.

Définition 2.6 (Distance) La distance entre deux nœuds distincts u et v , notée $\text{Dist}(u, v)$, est la longueur du plus court chemin de u à v .

Définition 2.7 (Diamètre) Le diamètre d'un graphe G est la plus grande distance séparant deux nœuds distincts de G .

Soit $\text{MaxDist}(u)$ la valeur maximale des distances $\text{Dist}(u, v)$ pour tous les nœuds v de G . Le diamètre de G est la valeur maximale des $\text{MaxDist}(u)$ pour tous les nœuds u de G .

Exemple 2.2 *Considérons le graphe G sur la figure 2.1. Le chemin (A, C, B, F, H) entre les nœuds A et H est un chemin de longueur 4. Il existe un autre chemin entre A et H , (A, B, E, H) de longueur 3. Donc, la distance entre les nœuds A et H est égale à 3. Le diamètre du graphe G est 3.*

Définition 2.8 (Composante connexe) *La composante connexe associée à un nœud u d'un graphe non orienté G est l'ensemble de tous les nœuds v de G tels qu'il existe un chemin de u à v .*

Un graphe non orienté G est dit *connexe* s'il existe un chemin entre tout couple de nœuds du graphe. De manière similaire, la *composante fortement connexe* associée à un nœud u de G est l'ensemble de tous les nœuds v de G tels qu'il existe un chemin orienté de u à v et un chemin orienté de v à u . Un graphe orienté G est dit *fortement connexe* s'il est réduit à une seule composante fortement connexe.

Exemple 2.3 *Sur le graphe G de la figure 2.1, $\{B, C, D, F\}$ est la composante fortement connexe associée au nœud B . Le graphe orienté G n'est pas fortement connexe car A et B n'ont pas la même composante fortement connexe. En effet, la composante fortement connexe du nœud A est $\{A\}$ alors que la composante fortement connexe du nœud B est $\{B, C, D, F\}$.*

2.2.2 Systèmes de transitions

Il est souvent commode de représenter un système qui changent d'état sous la forme d'un système de transition.

Définition 2.9 (Système de transition) *Un système de transition est un triplet $\mathcal{S} = (\mathcal{Q}, \mathcal{A}, \mathcal{T})$, où \mathcal{Q} est un alphabet fini représentant l'ensemble des états possibles du système, \mathcal{A} un alphabet fini représentant les actions possibles et $\mathcal{T} \subset \mathcal{Q} \times \mathcal{A} \times \mathcal{Q}$ l'ensemble des transitions possibles.*

Un système de transition est dit *déterministe* si et seulement si, à partir de n'importe quel état q , il n'existe pas deux transitions ayant pour origine q et portant la même action. Formellement, cela signifie qu'il n'existe pas deux transitions (q_1, a_1, q_2) et (q_3, a_2, q_4) dans \mathcal{T} telles que $q_1 = q_3$, $a_1 = a_2$ et $q_2 \neq q_4$.

Exemple 2.4 *Un système de transition peut être vu comme un graphe orienté où les arcs sont étiquetés par des actions. Un exemple de système de transition est présenté sur la figure 2.2. L'ensemble des états est $\mathcal{Q} = \{1, 2, 3, 4\}$ et l'ensemble des actions est $\mathcal{A} = \{a, b\}$. Les transitions sont représentées par des flèches étiquetées par les actions. L'état 1 est le seul état initial et l'état 4 le seul état terminal. Ce système de transition n'est pas déterministe car l'état 3 est origine de deux transitions portant la même action b .*

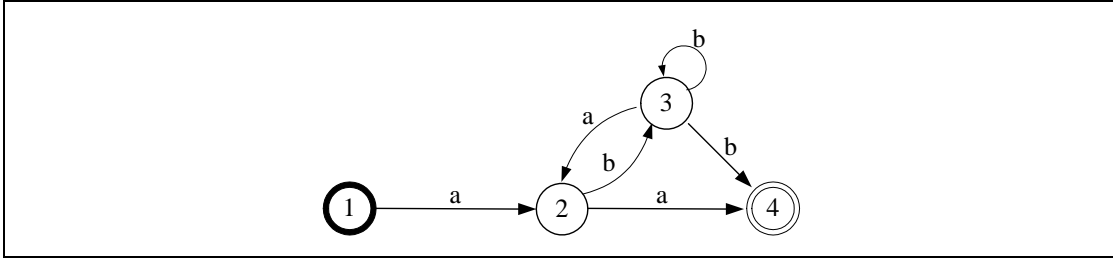


FIG. 2.2 – Un exemple de système de transition

Lorsqu'un système est composé de machines séquentielles sans lien entre elles, le comportement du système est décrit par la composition des systèmes de transitions locaux de chacune des machines du système. Cette composition génère un système de transition global dont l'ensemble des états est le produit cartésien des états des systèmes de transitions le composant. Nous associons l'action vide, notée $\{-\}$, à des transitions du type (q_i, a_j, q_i) dont l'exécution ne change pas l'état courant du système.

Définition 2.10 (Produit cartésien) *Le produit cartésien d'un ensemble de n systèmes de transitions $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_n\}$ où chaque $\mathcal{S}_i = (\mathcal{Q}_i, \mathcal{A}_i, \mathcal{T}_i)$, est un système de transitions $\mathcal{S} = (\mathcal{Q}, \mathcal{A}, \mathcal{T})$ où :*

- $\mathcal{Q} = \mathcal{Q}_1 \times \mathcal{Q}_2 \times \dots \times \mathcal{Q}_n$
- $\mathcal{A} = \prod_{1 \leq i \leq n} (\mathcal{A}_i \cup \{-\})$
- $\mathcal{T} = \{(q_1, q_2, \dots, q_n), (a_1, a_2, \dots, a_n), (q'_1, q'_2, \dots, q'_n)\}$, tel que $\forall i \in \{1, \dots, n\}$
 $(a_i = \{-\} \Rightarrow q_i = q'_i)$ ou $(a_i \neq \{-\} \Rightarrow (q_i, a_i, q'_i) \in \mathcal{T}_i)$

Exemple 2.5 *Un exemple de produit synchronisé entre deux systèmes de transitions \mathcal{S}_1 et \mathcal{S}_2 est présenté sur la figure 2.3. Le nombre d'états du système de transition global est le produit des nombres d'états des systèmes de transitions le composant, soit $3 \times 2 = 6$. Afin de simplifier la présentation, les transitions pour lesquelles tous les systèmes de transitions de base effectuent des actions vides ne sont pas représentées.*

À contrario, les composants de base d'un système réparti réagissent entre eux (les actions d'un composant peuvent influencer celles d'un autre). Pour rendre compte de ces interactions, on peut restreindre les comportements du système de transition en n'autorisant que certaines actions composées. On appelle *ensemble de synchronisation* la liste de ces actions autorisées pour le système de transition produit.

Définition 2.11 (Produit synchronisé) *Le produit synchronisé d'un ensemble de n systèmes de transitions $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_n\}$, où chaque $\mathcal{S}_i = (\mathcal{Q}_i, \mathcal{A}_i, \mathcal{T}_i)$, et avec l'ensemble de synchronisation $\mathcal{S}' \subset \prod_{1 \leq i \leq n} (\mathcal{A}_i \cup \{-\})$, est un système de transition $\mathcal{S} = (\mathcal{Q}, \mathcal{A}, \mathcal{T})$ où :*

- $\mathcal{Q} = \mathcal{Q}_1 \times \mathcal{Q}_2 \times \dots \times \mathcal{Q}_n$

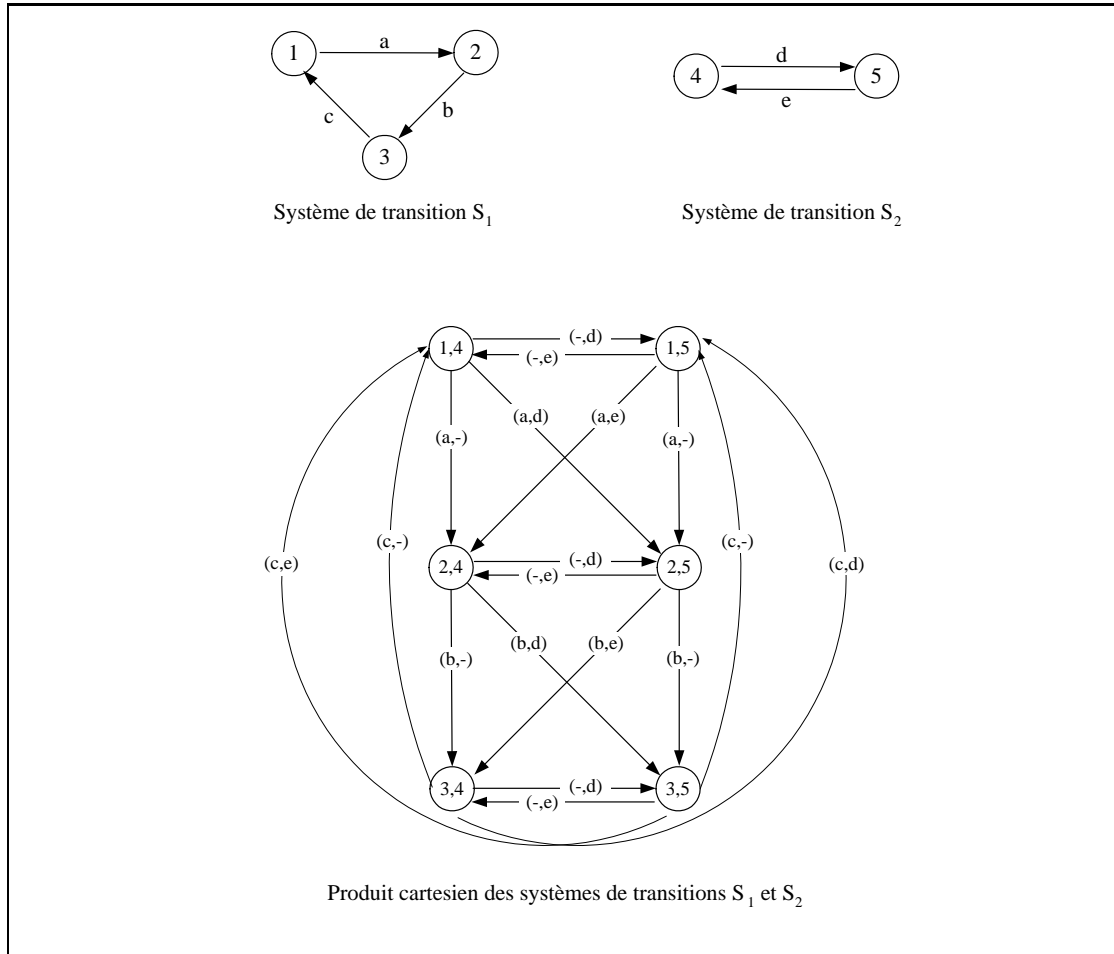
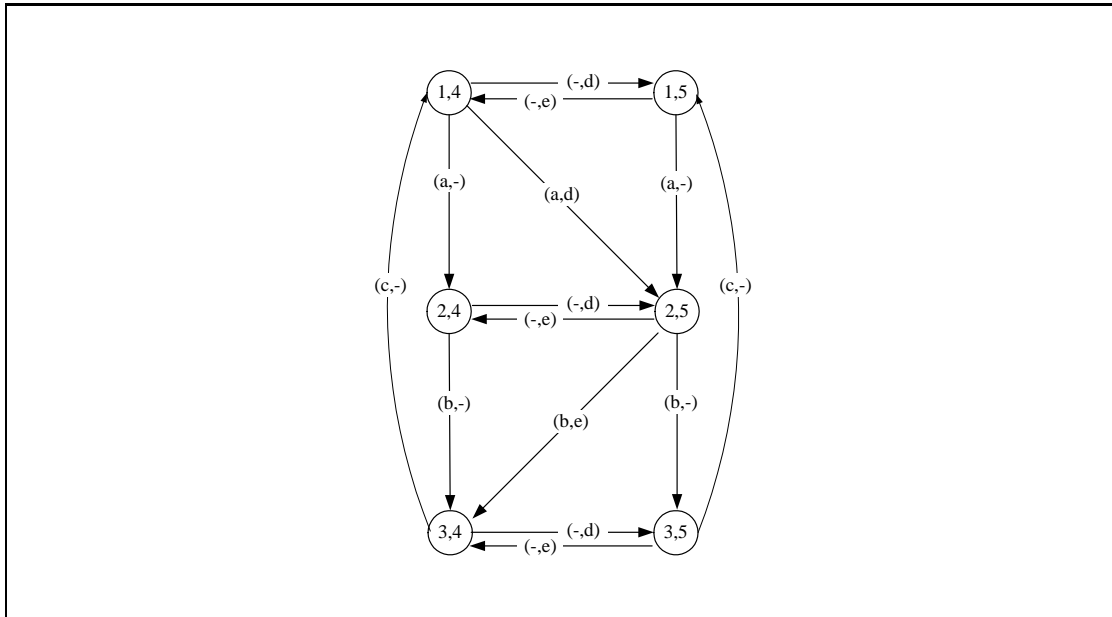


FIG. 2.3 – Produit cartésien de deux systèmes de transitions

- $\mathcal{A} = \mathcal{S}'$
- $\mathcal{T} = \{(q_1, q_2, \dots, q_n), (a_1, a_2, \dots, a_n), (q'_1, q'_2, \dots, q'_n)\}$, tel que $(a_1, a_2, \dots, a_n) \in \mathcal{A}$,
 $\forall i \in \{1, \dots, n\}, (a_i = \{-\} \Rightarrow q_i = q'_i)$ ou $(a_i \neq \{-\} \Rightarrow (q_i, a_i, q'_i) \in \mathcal{T}_i)$

Exemple 2.6 Un exemple de produit synchronisé entre les deux systèmes de transitions S_1 et S_2 est présenté sur la figure 2.4, avec l'ensemble de synchronisation \mathcal{S}' restreint aux transitions suivantes $\{(a, -), (b, -), (c, -), (-, e), (-, d), (a, d), (b, e)\}$. Les actions autorisées sont : soit a et b du système S_1 synchronisées avec respectivement d et e du système S_2 , soit seul l'un des deux systèmes effectue une action. Les états de ce système de transition sont obtenus par le produit cartésien de tous les états des systèmes de transitions synchronisés, soit $3 \times 2 = 6$.

FIG. 2.4 – Produit synchronisé des systèmes \mathcal{S}_1 et \mathcal{S}_2

2.3 Composants de base d'un système réparti

Les processeurs sont les éléments de calcul d'un système réparti. Pour mener à bien leurs calculs, ils utilisent des liens de communication qui leur permettent d'échanger des informations. Nous modélisons ces deux composants par des systèmes de transition, en tenant compte que les actions de communication des processeurs et des liens devront être synchronisées. Ces actions se divisent en trois ensembles disjoints :

- L : ensemble des actions de lecture. Elles seront synchronisées avec des actions de lecture des registres dans le cas d'un système à mémoire partagée, ou des actions de réception dans un canal dans le cas d'un système à passage de messages.
- I : ensemble des actions internes. Elles n'ont pas à être synchronisées avec les liens de communications.
- E : ensemble des actions d'écriture. Elles seront synchronisées avec des actions d'écriture dans un registre partagé dans le cas d'un système à mémoire partagée ou des actions d'émission de message dans un canal dans le cas d'un système à passage de messages.

2.3.1 Liens de communication

Les liens de communication sont les supports physiques de la communication. Nous distinguons deux types de liens :

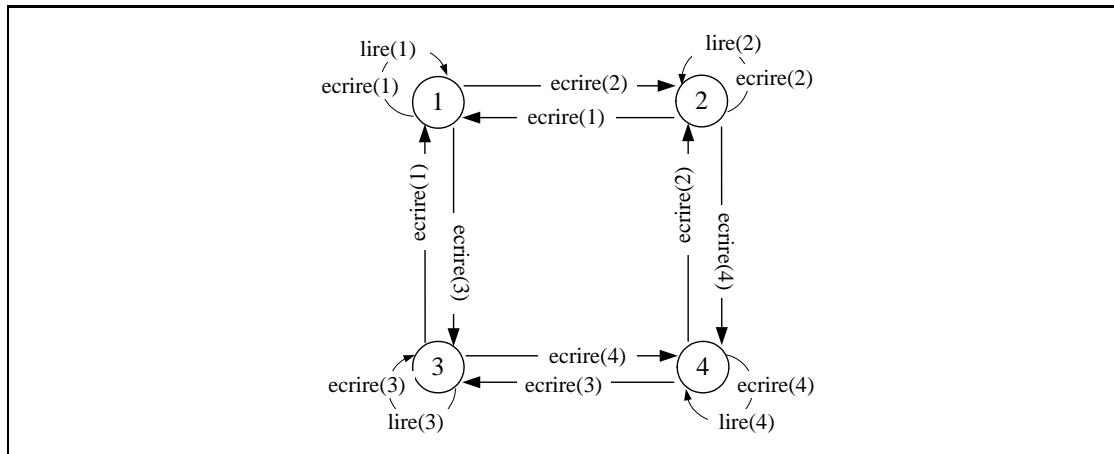


FIG. 2.5 – Exemple de registre

- les registres partagés : utilisés dans le modèle de communication par mémoire partagée, décrit à la sous-section 1.2.1, dans lequel deux processeurs voisins communiquent par des opérations de lecture et d'écriture effectuées sur deux registres partagés exclusivement entre eux. Un processeur dispose d'autant de registres que de voisins. Il peut lire et écrire dans ses propres registres, par contre il ne peut que lire les registres voisins. Un registre se modélise par un système de transition dont les états correspondent aux valeurs du registre partagé. Les actions sont de type lecture ou écriture et seront synchronisées avec les actions correspondantes au niveau des processeurs. L'action $ecrire(r, v)$ écrit la valeur v dans le registre r . Quant à l'action $lire(r, v)$, elle ne change pas l'état du système. Par ailleurs, chaque état du système de transition possède deux arcs bouclants étiquetés respectivement par $ecrire(r, v)$ et $lire(r, v)$. Un exemple de registre à quatre états est présenté sur la figure 2.5. Lorsqu'il n'y a qu'un seul registre, nous omettons le libellé des actions de lecture et d'écriture.
- les canaux : utilisés dans le modèle de communication par passage de message, décrit à la sous-section 1.2.1, dans lequel les processeurs communiquent par envoi et réception de messages. Les messages circulent entre l'expéditeur et le récepteur en traversant un canal reliant les deux processeurs. Un canal se modélise par un système de transition dont les états sont des multi-ensembles, appelés messages. Ses transitions sont de type émission ou réception et sont synchronisées avec les actions correspondantes au niveau des processeurs. L'action $emission(c, m)$ ajoute le message m sur le canal c et l'action $reception(c, m)$ retire le message m du canal c . Lorsqu'il n'y a qu'un seul canal, nous omettons le libellé des actions d'émission et de réception. Un exemple de canal manipulant deux messages m_1 et m_2 est présenté sur la figure 2.6. Les canaux peuvent être classés en plusieurs types selon

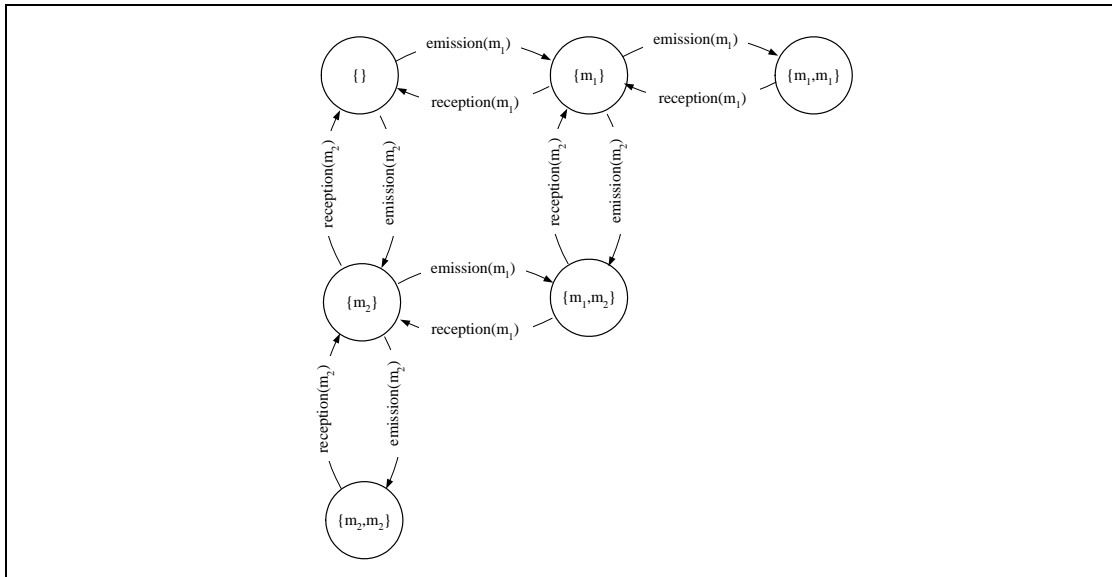


FIG. 2.6 – Exemple de canal

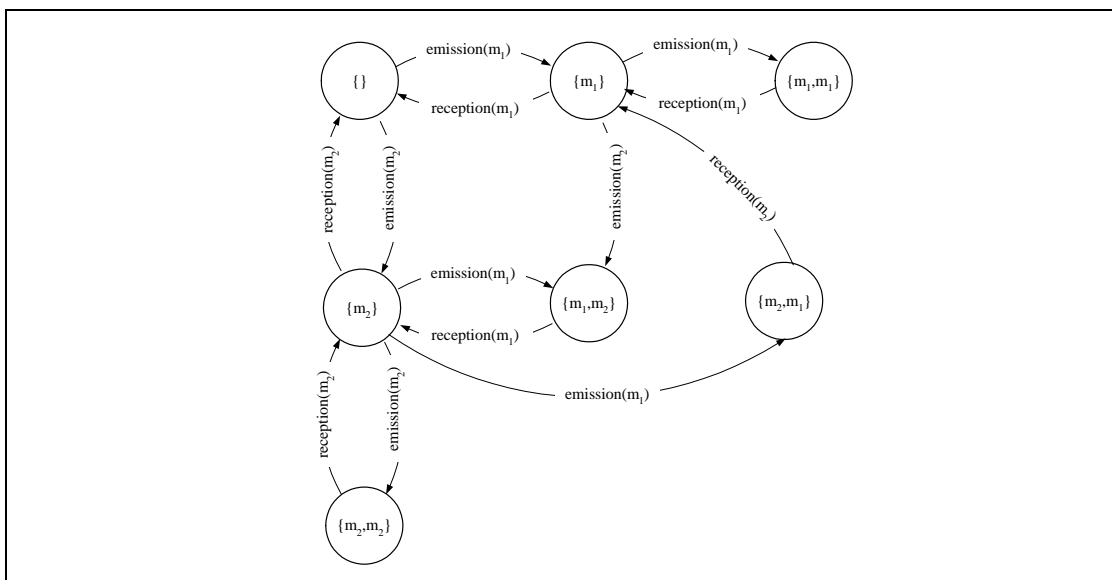


FIG. 2.7 – Exemple de canal FIFO

leurs propriétés. Tout d'abord, les canaux *bornés* ne contiennent qu'un nombre fini de messages. Les canaux *non bornés* peuvent contenir une infinité de messages : les systèmes de transitions associés sont infinis. De même, les canaux peuvent contenir des messages de taille fixe ou variable. Enfin, on peut classer les canaux suivant les propriétés d'ordre de réception de messages. Un canal *FIFO* (de l'anglais, First In First Out) assure que les messages sont reçus dans l'ordre de leur émission. Dans le cas contraire, les messages peuvent se perdre ou être reçus dans un ordre différent de celui de leur envoi. Le canal présenté sur la figure 2.7 est l'équivalent en FIFO du canal de la figure 2.6.

Précisons qu'un lien peut être indifféremment un registre ou un canal, même s'il existe une différence fondamentale entre eux : dans un canal, 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 d'un canal, alors que ce n'est pas le cas pour un registre.

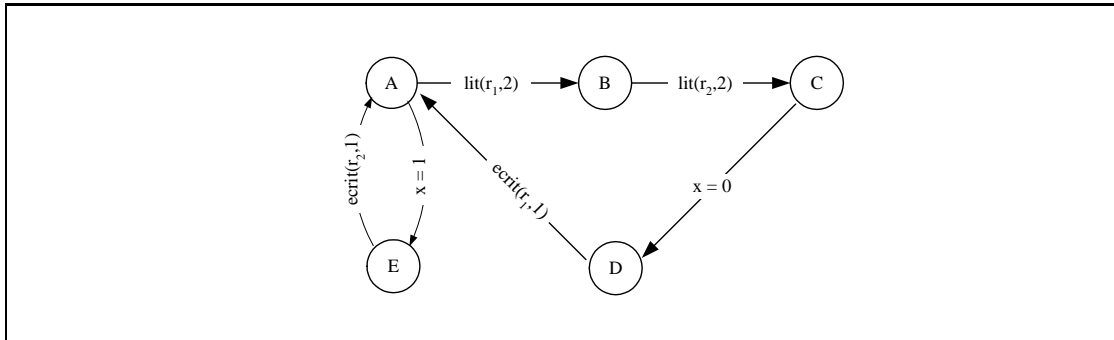
2.3.2 Processeurs

Les processeurs qui constituent le système peuvent être ou non différenciés les uns des autres. Un système est dit *uniforme* ou *anonyme* si les processeurs sont indifférenciés : ils sont tous identiques et exécutent le même algorithme. Il est dit *non-uniforme*, s'il devient possible de distinguer un processeur d'un autre. Un système est dit *semi-uniforme*, si on distingue un processeur particulier pour exécuter un algorithme différent des autres processeurs.

Une autre différence fondamentale des processeurs réside dans le déterminisme. Intuitivement, un *processeur déterministe* est un processeur dont la sortie ne dépend que de ses entrées. Un *processeur probabiliste* est un processeur qui peut prendre des décisions aléatoires à partir d'un même état. Cette différence est fondamentale car il existe des problèmes qui ne peuvent pas être résolus dans un cadre purement déterministe (i.e., le consensus dans un système asynchrone, Fischer, Lynch et Paterson, 1985).

Formellement, un processeur est un système de transition dont les états indiquent les différentes valeurs des variables utilisées et dont les transitions indiquent les différentes actions qu'il peut effectuer.

Définition 2.12 (Processeur) *Un processeur est un système de transition déterministe $\mathcal{S} = (\mathcal{Q}, \mathcal{A}, \mathcal{T})$, où \mathcal{Q} est l'alphabet constitué des états possibles du processeur, \mathcal{A} le produit cartésien des états possibles des voisins du processeur et \mathcal{T} la fonction de transition associée à l'algorithme du processeur.*

FIG. 2.8 – Système de transition des règles gardée R_1 et R_2

Les processeurs sont modélisés par des systèmes de transitions, mais décrits par des algorithmes locaux pour plus de lisibilité. Lorsqu'on écrit un algorithme, on utilise fréquemment des variables nommées qui prennent leurs valeurs dans un ensemble fini pour décrire l'état d'un processeur. Ce dernier est alors le produit cartésien des valeurs de chacune de ses variables. De façon générale, un algorithme réparti est décrit sous forme d'une disjonction finie de *règles gardées* de la forme : si la garde est vraie alors on effectue l'action correspondante. Chaque règle est identifiée par une étiquette.

Définition 2.13 (Règle gardée) Une règle gardée d'un processeur p est une règle de la forme *garde* \rightarrow *action*, où la partie gauche est appelée la garde et la partie droite l'action. La garde est un prédicat sur l'état du processeur p et sur celui de ses voisins. L'action est une affectation des variables de p .

Dans le modèle à états, pour évaluer la garde d'une règle gardée, un processeur p doit lire le contenu des toutes les variables apparaissant dans la garde. Une règle dont la garde est vide est dite *spontanée*. Une *règle activable* est une règle dont la garde est valide. Un *processeur activable* est un processeur dont l'une de ses règles est activable.

Définition 2.14 (Pas atomique) Un pas atomique d'un processeur p est la lecture de l'état de ses voisins suivie de la lecture de son état courant.

Exemple 2.7 Considérons un processeur dont le code est donné sous la forme de deux règles gardées R_1 et R_2 et qui peut communiquer au moyen de deux registres r_1 et r_2 .

$$\begin{aligned}
 R_1 : & \text{ lire}(r_1, 2) \wedge \text{ lire}(r_2, 2) \longrightarrow x = 0; \text{ ecrire}(r_1, 1); \\
 R_2 : & \longrightarrow x = 1; \text{ ecrire}(r_2, 1);
 \end{aligned}$$

Cet algorithme ne résout aucun problème. Il permet d'illustrer les différents types de gardes. La garde de la règle R_1 contient deux opérations de lecture alors que son action contient une action interne (l'affectation de 0 à x) et une action d'écriture. La règle R_2 est spontanée, son action contient une action interne (l'affectation de 1 à x) et une action d'écriture.

Pour passer d'une disjonction de règles gardées à un système de transition, on met en oeuvre la transformation suivante :

- les valeurs des variables de l'algorithme et son pointeur de programme déterminent les états du système de transition ;
- chaque transition du système de transition est étiquetée par l'une des actions figurant dans l'une des règles gardées. Un système de transition représentant le processeur dont le code est décrit par les deux règles R_1 et R_2 est présenté sur la figure 2.8.

2.4 Systèmes répartis

A partir d'un système réparti, il est possible de construire un graphe de communication G modélisant la relation "est capable de transmettre des informations à". Dans ce graphe, si les liens sont des registres partagés, alors un arc (i, j) est présent si et seulement si :

- une action $ecrire(v)$ sur r_{ij} fait partie de l'ensemble des actions du processeur p_i ,
- une action $lire(v)$ de r_{ij} fait partie de l'ensemble des actions du processeur p_j , et
- r_{ij} est un registre partagé exclusivement par p_i et p_j .

Définition 2.15 *Un système réparti \mathcal{S} est le produit synchronisé de l'ensemble des systèmes de transitions $P = \{P_1, \dots, P_n\}$ représentant les processeurs et de l'ensemble des systèmes de transitions $L = \{L_1, \dots, L_m\}$ représentant les liens de communication.*

L'état global d'un système réparti à un instant précis est entièrement décrit par une configuration.

Définition 2.16 (Configuration) *Une configuration d'un système réparti $\mathcal{S} = (P, L)$, est un état du produit synchronisé des systèmes de transitions P et L . Par construction, une configuration c de \mathcal{S} est caractérisée par un vecteur des états de tous les systèmes de transitions.*

L'ensemble des configurations d'un système \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 système de transition global. Les états sont les configurations du système, sa fonction de transition est la composition d'une ou de plusieurs actions et son ensemble de synchronisation contient les actions autorisées suivantes :

- exactement une action interne d'un processeur et l'action vide pour tous les autres systèmes de transitions ;
- une action de lecture pour un processeur, une action de lecture pour le lien associé et l'action vide pour tous les autres systèmes de transitions ;

- une action d'écriture pour un processeur, une action d'écriture pour le lien associé et l'action vide pour tous les autres systèmes de transitions.

Exemple 2.8 *Considérons un système réparti constitué de deux processeurs p_1 et p_2 et d'un registre r partagé par ces deux processeurs tel que :*

- le processeur p_1 dispose d'une unique règle $R_1 : lire(1) \longrightarrow ecrire(1)$,
- le processeur p_2 d'une unique règle $R_2 : lire(2) \longrightarrow ecrire(2)$,
- le registre r peut prendre l'une des deux valeurs entières 1 et 2.

Les systèmes de transitions correspondants aux processeurs p_1 , p_2 et au registre r , ainsi que le produit synchronisé de ses trois systèmes, sont donnés sur la figure 2.9. L'ensemble de synchronisation $\mathcal{S}' = \{(lire(1), -, lire(1)); (-, lire(2), lire(2)); (ecrire(1), -, ecrire(1)); (-, ecrire(2), ecrire(2))\}$. Pour la lisibilité du produit synchronisé, une transition est étiquetée par $lire(1)$ pour figurer l'action $(lire(1), -, lire(1))$ de l'alphabet de synchronisation.

Définition 2.17 (Transition) *Une transition d'un système réparti \mathcal{S} est l'application d'un ou de plusieurs pas atomiques par sous-ensemble non vide des processeurs du système. Tous les processeurs de ce sous-ensemble lisent leur état et l'état de leur voisin, puis écrivent simultanément leur nouvel état.*

Afin de dissocier le système de transition d'un processeur de celui du système réparti, nous utiliserons les termes *configuration* et *transition* pour décrire le comportement du système réparti, et les termes *état* et *pas atomique* pour décrire le comportement d'un processeur.

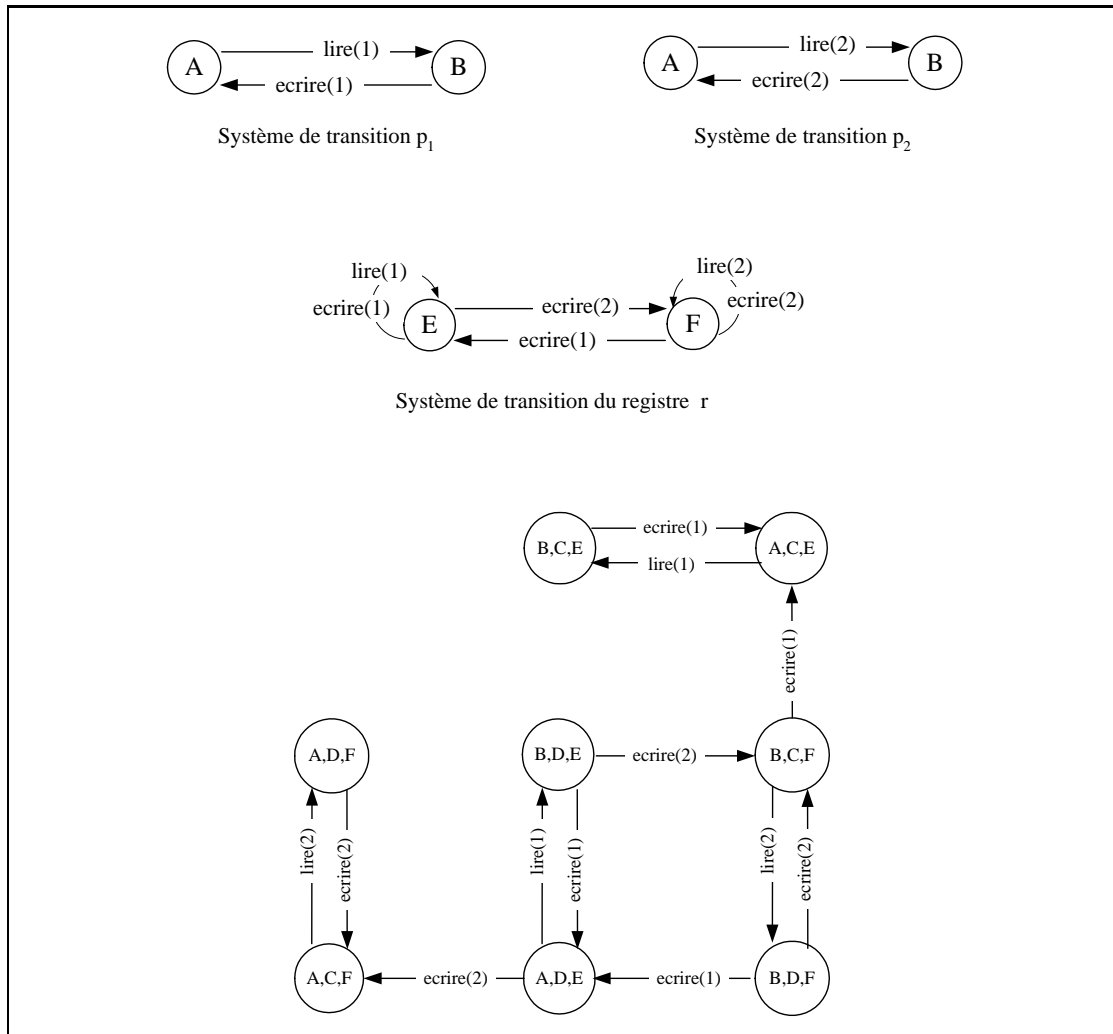
2.4.1 Exécutions

Une exécution d'un système réparti est une séquence de configurations qui se succèdent et évoluent suite à des événements séquentiels ou simultanés engendrés par les exécutions des différents processeurs.

Définition 2.18 (Exécution) *Une exécution d'un système réparti \mathcal{S} est une séquence maximale de configurations et d'actions de \mathcal{S} , notée $e = (c_1, a_1, c_2, a_2, \dots)$ où c_{i+1} est la configuration atteinte depuis c_i par l'exécution de l'action a_i . La configuration c_1 est appelée configuration initiale de e .*

Une exécution est soit finie, soit infinie mais aucune action de \mathcal{S} n'est possible à partir de 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} .

Il est souvent pratique de projeter les exécutions sur des espaces plus restreints pour caractériser leurs propriétés. Nous présentons maintenant trois types de projections :

FIG. 2.9 – Produit synchronisé de p_1 , p_2 et r

- les *facteurs* qui définissent de manière commode les propriétés portant sur une portion d'une exécution,
- les *journaux* qui décrivent les propriétés portant sur les configurations,
- les *traces* qui portent sur l'ordre dans lequel les actions apparaissent.

Définition 2.19 (Facteur) *Un facteur d'une exécution $e \in \mathcal{E}$ est une sous-séquence de e , dont le premier terme (éventuellement le dernier si le facteur est fini) est une configuration.*

Soit $e = c_1 a_1 c_2 a_2 c_3 a_3 \dots$ une exécution dont la configuration initiale est c_2 et la configuration finale est c_3 , alors $f = c_2 a_2 c_3$ est un facteur fini de e . Le facteur $f' = c_3 a_3 \dots$ est un facteur infini 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 noté \mathcal{F}_c . L'ensemble

des facteurs finis de \mathcal{S} dont la configuration initiale est α et la configuration finale est β est noté $\mathcal{F}_{\alpha,\beta}$. Une configuration $\beta \in \mathcal{C}$ est dite *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 $\alpha \rightsquigarrow \beta$.

Définition 2.20 (Journal) *Le journal d'une exécution est sa projection sur les configurations qui la constituent.*

Considérons l'exécution $e = c_1 a_1 c_2 a_2 c_3 a_3 \dots$. Le journal correspondant est $j = c_1 c_2 c_3 \dots$ dont la configuration initiale 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 fait partie d'un ensemble $C_1 \subset \mathcal{C}$ est noté \mathcal{J}_{C_1} .

Définition 2.21 (Trace) *La trace d'une exécution est sa projection sur les actions qui la constituent.*

Considérons l'exécution $e = c_1 a_1 c_2 a_2 c_3 a_3 \dots$. La trace correspondante est $t = a_1 a_2 a_3 \dots$ dont l'action initiale est a_1 . L'ensemble des traces de \mathcal{S} est noté \mathcal{T} . L'ensemble des traces de \mathcal{S} dont la configuration initiale est a est noté \mathcal{T}_a .

Définition 2.22 (Trace restreinte) *La trace restreinte $t_{\{a_1, \dots, a_n\}}$ d'une exécution est sa projection sur les actions $\{a_1, \dots, a_n\}$ qui la constituent.*

Considérons l'exécution $e = c_1 a_1 c_2 a_2 c_3 a_3 \dots$. La trace $t_{\{a_1, a_3\}} = a_1 a_3 \dots$ est la trace restreinte sur les actions a_1 et a_3 correspondante à l'exécution précédente et dont l'action initiale est a_1 .

2.4.2 Atomicité

C'est une propriété liée aux exécutions d'un système modélisé par un système de transition. Chaque exécution est vue comme une suite de *pas atomiques*. Il existe plusieurs définitions de la notion d'atomicité. Une des ses définitions consiste à considérer un pas atomique (ou *action atomique*) comme la plus petite action que le système peut exécuter sans être interrompu. Cette définition est celle de Dolev, Israeli et Moran (1993), qui répertorie aussi deux niveaux d'atomicités :

- l'atomicité composite, qui considère qu'un pas atomique est composée de plusieurs actions d'entrée (i.e., la lecture d'un registre ou la réception d'un message), de plusieurs actions internes et d'une action de sortie (i.e., l'écriture sur un registre ou l'émission d'un message),
- l'atomicité lecture/écriture, qui considère qu'un pas atomique est composée de plusieurs actions internes et d'une seule action d'entrée ou de sortie.

Notons que l'atomicité lecture/écriture est le grain d'atomicité le plus fin utilisé en pratique. En effet, une atomicité plus fine serait envisageable (en considérant que toutes les actions sont atomiques, qu'elles soit internes ou de communication), mais elle rendrait les preuves très difficiles.

2.4.3 Propriété d'équité

L'équité est une propriété en rapport avec la *nature* des actions rencontrées au cours des exécutions infinies d'un système réparti. Elle garantit qu'une action ou un processeur ne soit pas défavorisé par rapport aux autres.

Définition 2.23 (équité faible) *Une exécution e est dite faiblement équitable si, à tout moment au cours de cette exécution, il n'existe pas de processeur activable dans toutes les configurations suivantes sans plus jamais être activé.*

Définition 2.24 (équité forte) *Une exécution e est dite fortement équitable si, à tout moment au cours de cette exécution, il n'existe pas de processeur qui soit infiniment souvent activable par la suite sans plus jamais être activé.*

Autrement dit, une exécution est faiblement équitable si et seulement si, au long de cette exécution, tout processeur qui reste activable dans une infinité de configurations consécutives effectue infiniment souvent une action. Une exécution est fortement équitable si et seulement si, au long de cette exécution, tout processeur qui est infiniment souvent activable effectue infiniment souvent une action.

Par ailleurs, une exécution e est dite *équitable* si sa trace contient infiniment souvent des actions de tous les processeurs. Autrement dit, tous les processeurs sont infiniment souvent activables, et donc les actions correspondantes sont exécutées au bout d'un temps fini.

Exemple 2.9 *Considérons un système à deux processeurs, numérotés P_1 et P_2 , et la liste des processeurs activables $(\{P_1, P_2\}, \{P_1\})^*$ (i.e., $\{P_1, P_2\}$ sont activables, puis $\{P_1\}$, puis $\{P_1, P_2\}$, puis $\{P_1\}$, etc...). Une exécution de ce système est faiblement équitable si le processeur P_1 est infiniment souvent sélectionné, et elle est fortement équitable si les deux processeurs sont infiniment souvent sélectionnés.*

2.4.4 Démon

Pour décrire le comportement d'un système, il ne suffit pas de connaître les règles qui définissent les actions des processeurs, il faut également savoir dans quel ordre ces actions vont être effectuées. Une transition est donc caractérisée par le choix des processeurs qui vont effectuer une action et par le choix des règles qui s'appliquent à ces processeurs. Pour définir cet ordonnancement des actions, un mécanisme extérieur, appelé *démon*, a

été introduit. Un démon est un *ordonnanceur* (de l'anglais *scheduler*) qui choisit l'ordre suivant lequel les processeurs vont effectuer leurs actions. Il sélectionne, pour chaque étape d'une exécution, le ou les processeurs qui vont effectuer une action. Bien évidemment, le démon ne peut pas choisir à chaque étape n'importe quel processeur. Il choisit parmi ceux qui sont activables (i.e., qui peuvent effectuer une action). Plus précisément :

Définition 2.25 (Démon) *Un démon \mathcal{D} est un mécanisme qui, à chaque état d'une exécution, choisit un sous-ensemble non vide de processeurs activables, pour lesquels une règle va s'appliquer.*

Le choix de ce sous-ensemble peut se faire de manière déterministe ou probabiliste selon le démon considéré. De même, le mécanisme de choix peut prendre en compte tout l'historique de l'exécution ou uniquement l'état courant. Le démon peut également choisir *a priori* un ou plusieurs processeurs parmi ceux qui sont activables. Il peut donc être considéré comme un mécanisme limitant l'ensemble des exécutions possibles d'un système.

Plus qu'un ordonnanceur, un démon peut aussi être vu comme un *adversaire* ayant un certain pouvoir et capable de mettre le système dans n'importe quel état. Son pouvoir réside dans le fait qu'il peut privilégier certains processeurs et empêcher d'autres d'agir, ou encore qu'il peut retenir et corrompre des messages. Son but est de compliquer la tâche au système, en agissant de manière à rendre son exécution difficile et à l'empêcher d'effectuer la tâche pour laquelle il est conçu.

Cette définition d'un démon est peu restrictive, et permet de représenter une large gamme de comportements. Dans la littérature, nous retrouvons un grand nombre de démons classés selon les propriétés du système et de l'environnement dans lequel ils évoluent. Nous ne citerons que les démons utilisés dans les systèmes à mémoire partagée :

- Le démon *lecture/écriture*, noté \mathcal{D}_{LE} , considère que seules les actions de lecture ou d'écriture sont atomiques. A part cela, aucune contrainte n'est imposée quant à l'ordre d'exécution des actions.
- Le démon *totalelement 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 que les règles sont évaluées atomiquement et que les actions associées sont exécutées atomiquement. Formellement, un processeur est autorisé à lire l'ensemble des variables de ses voisins ou à écrire dans toutes ses propres variables en une seule action atomique.
- Le démon *réparti*, noté \mathcal{D}_R , est un démon tel qu'une action atomique est une séquence de phases d'évaluation des gardes, suivie d'une phase d'exécution des actions

associées. Formellement, le démon choisit un sous-ensemble \mathcal{P} de processeurs parmi ceux qui peuvent agir. Tous les processeurs de \mathcal{P} doivent simultanément et en une seule action atomique lire les variables de leurs voisins, et ensuite écrire dans leurs propres variables.

- Le démon *centralisé*, noté \mathcal{D}_C , est un démon réparti tel qu’une action atomique est une séquence de phases impliquant un unique processeur du système. Formellement, le sous-ensemble \mathcal{P} ne doit contenir qu’un seul processeur. Ce dernier doit en une action atomique lire les variables de ses voisins et ensuite écrire dans ses propres variables.
- Le démon *synchrone*, noté \mathcal{D}_S , est un démon réparti tel qu’une action atomique est une séquence de phases impliquant tous les processeurs activables du système. Formellement, \mathcal{P} doit contenir tous les processeurs pouvant agir. Tous les processeurs de \mathcal{P} doivent simultanément et en une seule action atomique lire les variables de leurs voisins et ensuite écrire dans leurs propres variables.

Afin d’établir une hiérarchie entre les démons, Tixeuil (2000) définit la relation *plus puissant* suivante : soit un système \mathcal{S} avec \mathcal{E}_A (resp. \mathcal{E}_B) l’ensemble d’exécutions de \mathcal{S} sous le démon A (resp. B), le démon A est plus puissant que le démon B si et seulement si $(\mathcal{E}_B \subset \mathcal{E}_A) \wedge (\mathcal{E}_B \neq \mathcal{E}_A)$.

Il résulte de cette hiérarchie 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.

A cela vient s’ajouter des propriétés sur la manière de choisir l’ensemble \mathcal{P} des processeurs autorisés à agir :

- Un démon *non-équitable* est totalement libre. Si un processeur peut continuellement agir, alors le démon peut le choisir éternellement, privant ainsi les autres processeurs du système de l’exécution de leurs actions.
- Un démon *équitable* interdit qu’un processeur pouvant agir soit systématiquement exclu de \mathcal{P} . Toutes les exécutions qu’il engendre vérifient la propriété d’équité forte.
- Un démon *probabiliste* choisit l’ensemble \mathcal{P} de façon aléatoire, généralement en fonction d’une loi de probabilité.

2.5 Auto-stabilisation

Dans les systèmes répartis non stabilisants, l'ensemble des exécutions est fortement restreint par les contraintes imposées aux configurations initiales. Les systèmes auto-stabilisants sont moins restrictifs puisque toutes les configurations du système sont potentiellement des configurations initiales. Formellement, nous définissons les systèmes répartis auto-stabilisants par les trois propriétés de correction, de convergence et de clôture qui sont décrites ci-après.

Définition 2.26 (Auto-stabilisation) *Un système \mathcal{S} est auto-stabilisant pour la spécification \mathcal{P} s'il existe un sous-ensemble \mathcal{L} de l'ensemble des configurations \mathcal{C} vérifiant :*

- toute exécution de \mathcal{S} dont la configuration initiale appartient à \mathcal{L} vérifie la spécification \mathcal{P} (correction),
- toute exécution de \mathcal{S} atteint une configuration de \mathcal{L} (convergence),
- toute exécution de \mathcal{S} dont la configuration initiale appartient à \mathcal{L} n'atteint aucune configuration illégitime (clôture).

Dans la suite nous détaillons ces trois propriétés fondamentales des algorithmes répartis auto-stabilisants.

2.5.1 Correction

Le concept de correction est couramment utilisé dans l'algorithmique répartie. Il est la garantie que toute exécution ayant une configuration initiale légitime satisfait la *spécification* du problème. Formellement, une spécification est une famille d'ensemble d'exécutions possédant une propriété commune. Chaque ensemble d'exécutions se rapporte à un système résolvant le problème spécifié.

Définition 2.27 (Spécification) *La spécification d'un problème est un prédicat portant sur les exécutions du système.*

Étant donné une spécification \mathcal{P} d'un problème, la vérification d'une spécification consiste à savoir si un système repartit \mathcal{S} résout ce problème. Autrement dit, l'ensemble des exécutions d'un système \mathcal{S} vérifiant sa spécification \mathcal{P} est un sous ensemble $\mathcal{E}_{\mathcal{P}}$ de l'ensemble des exécutions \mathcal{E} tel que pour chaque journal j d'une exécution $e \in \mathcal{E}_{\mathcal{P}}$, toutes les configurations de j satisfont \mathcal{P} .

Exemple 2.10 *Considérons un système constitué de n processeurs. La spécification du problème de l'exclusion mutuelle est caractérisée par les deux prédicats suivant :*

- sûreté : au plus un processeur accède à la ressource ;
- vivacité : le temps d'attente d'un processeur pour accéder à la ressource est fini.

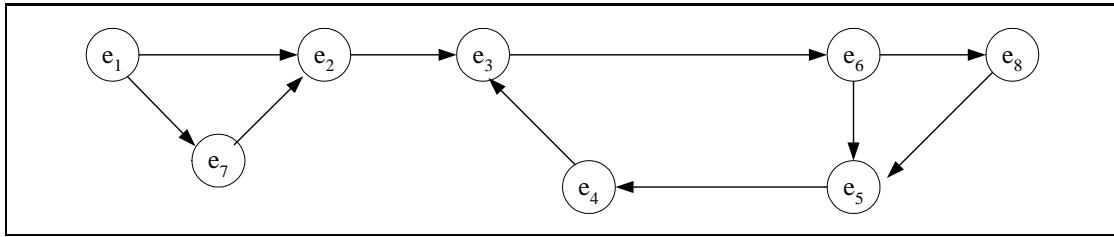


FIG. 2.10 – Exemple de convergence faible et de convergence forte

Dès lors, une exécution vérifie cette spécification si dans toutes les configurations un seul processeur accède à la ressource, et si à partir de toute configuration un processeur souhaitant accéder à la ressource y accédera au bout d'un temps fini.

Prouver la correction d'un algorithme auto-stabilisant ne diffère pas des preuves de correction des algorithmes répartis classiques. Partant d'un ensemble de configurations, il faut prouver que toute exécution effectuée à partir de ces configurations satisfait la spécification de la tâche du système. Les techniques habituelles d'invariants, qui consistent à exhiber une expression qui reste vraie quelle que soit l'exécution, peuvent être utilisées. Pour cela, on suppose qu'une exécution a une configuration initiale légitime ℓ . Ensuite, on montre que toutes les exécutions partant de ℓ vérifient la spécification du problème. Cela prouve la correction de l'algorithme.

2.5.2 Convergence

Explicitons tout d'abord la notion de convergence dans le cadre général des systèmes de transitions. Soit un système de transition $\mathcal{S} = (\mathcal{Q}, \mathcal{A}, \mathcal{T})$ et \mathcal{L} un sous ensemble de \mathcal{Q} . On dit que \mathcal{S} converge faiblement si à partir de tout état atteint lors d'une exécution, il existe une exécution permettant d'atteindre un état de \mathcal{L} . On dit que \mathcal{S} converge fortement si toute exécution atteint un état de \mathcal{L} . Dans les systèmes auto-stabilisants, \mathcal{L} est référencé comme l'ensemble des configurations légitimes.

Exemple 2.11 *Considérons le système de transition à 8 états présenté sur la figure 2.10. Le système converge faiblement vers l'ensemble $\{e_8\}$ et fortement vers l'ensemble $\{e_3, e_4, e_5, e_6\}$.*

La convergence est la propriété clé des systèmes auto-stabilisants : c'est l'assurance que toute exécution du système atteindra une configuration légitime. Pour prouver la convergence d'un système, il faut montrer qu'à partir de chacune de ses configurations, toute exécution mène à une configuration légitime. Les preuves de convergence des algorithmes auto-stabilisants sont souvent difficiles et ont fait l'objet de nombreuses études. Nous présentons ici les deux techniques les plus utilisées : la preuve par mesure décroissante et la preuve par attracteur.

Mesure décroissante. Cette technique a été formalisée par Gouda (1995). Elle consiste à exhiber une mesure à valeurs entières positives sur chaque configuration du système. Cette mesure doit en outre être minimale pour les états légitimes. L'approche consiste à prouver que cette mesure décroît 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 mesure garantit qu'à partir de toute configuration, et pour toute exécution possible à partir de cette configuration, le système finit par atteindre une configuration légitime. L'inconvénient de cette technique réside dans son manque de généralité. En effet, le choix d'une mesure se base sur les valeurs possibles des différentes variables manipulées par un algorithme ainsi que sur leur évolution les unes par rapport aux autres au cours du temps. Il en résulte que les mesures dépendent très fortement de l'algorithme utilisé et ne peuvent donc pas être réutilisées pour un autre algorithme.

Attracteurs. Gouda et Multari (1991) ont défini une autre technique, souvent utilisée en complément des mesures décroissantes, et qui consiste à montrer que le système converge successivement d'un ensemble de configurations vers un autre, plus petit. Intuitivement, les attracteurs sont des ensembles de configurations tels qu'à partir de toute configuration d'un ensemble donné, toute exécution atteint nécessairement une configuration de l'attracteur. Considérons un système \mathcal{S} dont l'ensemble des configurations est \mathcal{C} . Prouver la convergence du système revient alors à trouver des ensembles de configurations $\mathcal{C}_1, \dots, \mathcal{C}_n$ tels que pour tout $i \in \{2, \dots, n\}$, \mathcal{C}_i est un attracteur de \mathcal{C}_{i-1} et tels que $\mathcal{C}_1 = \mathcal{C}$ et \mathcal{C}_n est l'ensemble des configurations légitimes. En quelque sorte, les attracteurs permettent de définir des paliers de convergence, chaque palier regroupant plusieurs états du système ayant les mêmes propriétés. La convergence consiste à atteindre le palier dont les propriétés caractérisent les états légitimes. L'inconvénient de cette décomposition des états du système en plusieurs paliers (ou *modules*) est qu'à aucun moment elle ne peut être automatisée.

Face aux inconvénients des méthodes classiques, plusieurs approches se sont développées pour la simplification et l'automatisation des preuves :

- une technique de simplification des preuves consiste à contraindre de manière automatique l'environnement pour qu'il fournisse des exécutions ayant des propriétés particulières. Ceci est réalisé grâce aux démons. En effet, les mesures sont liées à l'évolution des valeurs des différentes variables. L'utilisation d'un démon faible, comme le démon centralisé, simplifie les preuves en fournissant un cadre où les actions des processeurs sont traitées séquentiellement. Or en réalité, les démons qui répondent le mieux aux exigences informatiques des systèmes répartis sont les démons forts, comme les démons répartis. Par conséquent, il est intéressant de trouver des moyens automatiques de transformation pour

qu'un algorithme conçu sous un démon faible puisse fonctionner pour la même spécification sous un démon fort. Cette transformation est réalisée par la technique de *composition* d'algorithmes. L'idée de composer des algorithmes auto-stabilisants afin de pouvoir renforcer leur adaptabilité a été introduite par Gouda et Herman (1991). Intuitivement, un algorithme est composé de plusieurs couches notées de 1 à k , telle que la couche i (pour $1 < i \leq k$) dépend des variables qui se stabilisent grâce aux actions des couches 1 à $i - 1$. Ils ont également introduit la *composition par sélection*. Les deux modules composés sont indépendants mais peuvent modifier les mêmes variables de sorties. La sélection se fait par un prédicat qui vaut vrai uniquement pour le module qui a le droit de modifier les variables de sorties et faux pour l'autre. Plus tard, Varghese (1997) a introduit la *composition des modules indépendants*. Les composants interagissent entre eux à l'aide de leurs sorties. Les propriétés de l'algorithme composé sont obtenues en conjugant les propriétés des modules composants. Plus récemment, Beauquier, Gradinariu et Johnen (1999c) ont défini la *composition croisée* conçue afin de fortifier un algorithme auto-stabilisant dans son interaction avec le démon. A l'aide d'un algorithme B dont les exécutions vérifient certaines propriétés (comme l'équité) sous un démon puissant, la composition croisée consiste à gérer l'influence du démon sur l'algorithme A , appelé *module faible*, à travers les propriétés de l'algorithme B , appelé *module fort*. Ainsi, l'algorithme produit A' est auto-stabilisant pour la même spécification que A mais sous un démon plus puissant.

- une autre technique pour simplifier les preuves consiste à les automatiser. Nous avons vu que les méthodes basées sur des mesures sont très difficiles à automatiser à cause du manque de généralité des mesures. Il n'en est pas de même pour les *méthodes énumératives*. En effet, une manière standard de montrer des propriétés sur des exécutions est d'énumérer les configurations atteignables. Il existe de nombreux outils qui procèdent à ce genre d'approche, tels que le *model-checker*. Récemment, Beauquier, Bérard, Fribourg et Magniette (2001) ont mis au point une technique de réécriture, plus particulièrement les *surréductions*, permettant de prouver la convergence des algorithmes auto-stabilisants sur des anneaux. Cette technique permet de construire un graphe de transitions fini représentant le système, en restreignant les exécutions à considérer. Si toutes les exécutions sur ce graphe convergent vers l'ensemble voulu, alors l'algorithme original va lui aussi converger.

2.5.3 Clôture

D'après la définition, un système n'est auto-stabilisant que pour une spécification particulière. Lorsque cette spécification est de type "on reste toujours dans un ensemble de configurations E ", l'ensemble E est dit *clos*.

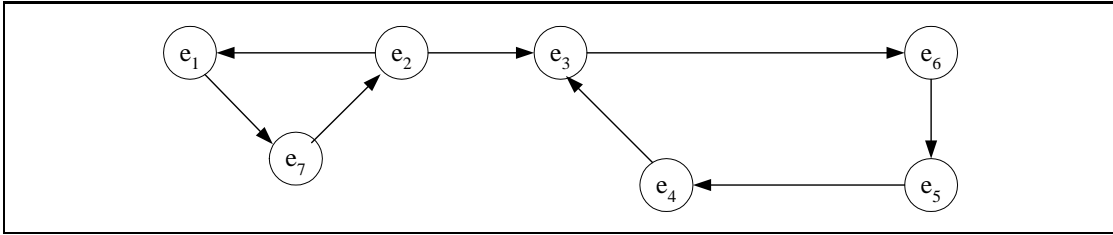


FIG. 2.11 – Exemple de clôture faible et de clôture forte

Définition 2.28 (Clôture) *Considérons un ensemble E et une relation de transitions T . On dit que E est clos pour T si pour tout élément e de E , (e, a, e') implique que $e' \in E$.*

Considérons un système de transition $\mathcal{S} = (\mathcal{Q}, \mathcal{A}, T)$. On dit que \mathcal{S} est *faiblement clos* si pour toute exécution à partir de E , il existe un suffixe où chaque état atteint est dans E . On dit que \mathcal{S} est *fortement clos* si chaque transition à partir d'un état de E atteint un état de E .

Exemple 2.12 *Considérons le système de transition à 7 états présenté sur la figure 2.11. Le système converge faiblement vers l'ensemble $\{e_1, e_2, e_7\}$ et fortement vers l'ensemble $\{e_3, e_4, e_5, e_6\}$.*

La notion de clôture est particulière au domaine de l'algorithmique auto-stabilisante. Elle exprime le fait que l'ensemble des configurations légitimes décrit l'intégralité du comportement du système une fois la convergence obtenue. Pour montrer l'auto-stabilisation d'un système \mathcal{S} vers une spécification \mathcal{P} , il faut montrer la convergence de \mathcal{S} vers E , et vérifier la clôture de E . Dans le cas où l'ensemble des configurations $E = \mathcal{L}$, l'ensemble des configurations légitimes, nous dirons que le système \mathcal{S} est auto-stabilisant vers $\mathcal{L} = E$ et non plus vers une spécification \mathcal{P} qui qualifie l'ensemble des exécutions qui restent dans E . Soit \mathcal{L} un ensemble de configuration légitime, \mathcal{L} est dit clos si toute configuration accessible à partir d'une configuration c de \mathcal{L} est, elle-même, une configuration légitime.

Nous illustrons la différence entre convergence et auto-stabilisation dans l'exemple suivant.

Exemple 2.13 *Considérons le système de transition à 7 états présenté sur la figure 2.12. Soient $L_1 = \{e_4, e_5\}$ et $L_2 = \{e_3, e_4, e_5, e_6\}$, deux sous-ensembles de l'ensemble des états du système. Par construction, L_1 est inclus dans L_2 . Le système converge vers l'ensemble L_1 car toute exécution infinie, quel que soit son état de départ, va passer par l'ensemble L_1 . Par contre le système n'est pas auto-stabilisant vers L_1 puisqu'il existe une exécution infinie $e_1, e_2, e_3, e_6, e_5, e_4, e_3, e_6, e_5, \dots$ qui est infiniment souvent hors de L_1 qui itère le cycle e_5, e_4, e_3, e_6 . Par conséquent, L_1 n'est pas clos (ni faiblement ni fortement). Par contre, du fait que l'ensemble L_2 contient L_1 (donc le système converge vers L_2) et que L_2 est fortement clos, le système de transition est auto-stabilisant vers L_2 . En résumé, le système converge vers L_1 mais s'auto-stabilise en L_2 .*

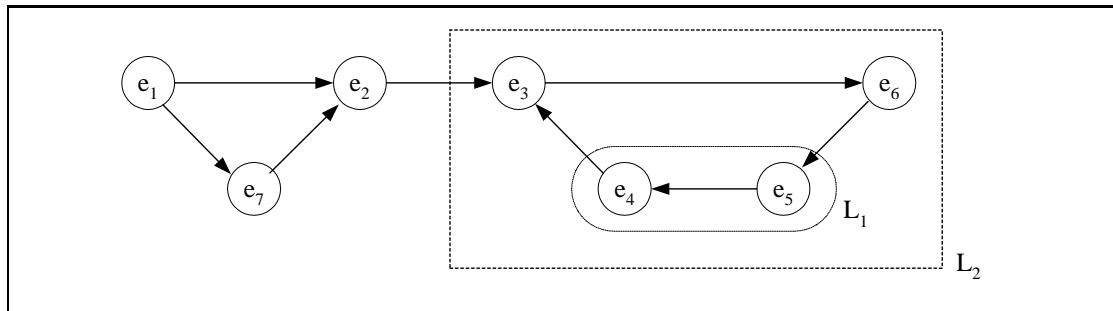


FIG. 2.12 – Un système de transition convergeant vers L_1 et auto-stabilisant vers L_2

2.6 Mesures de complexité

Diverses mesures de complexité sont utilisées pour comparer les performances des algorithmes auto-stabilisants. Ces mesures s'expriment en espace ou en temps.

Dans le modèle à états, la *complexité en espace* d'un algorithme se définit comme la taille en nombre de bits des variables utilisées par chaque processeur du système. Dans les modèles acceptant le passage de messages, on considère séparément la taille des messages et la quantité totale qui circule sur le réseau afin d'évaluer les ressources en bande passante utilisées par cet algorithme.

La *complexité en temps* est plus délicate à définir. Elle s'exprime de façon naturelle dans les systèmes synchrones : la mesure couramment admise est le nombre de phases synchrones. Dans le cadre de l'algorithmique asynchrone, plusieurs mesures ont été proposés, parmi lesquelles : le nombre de *pas atomiques* ou le nombre de *rondes asynchrones*. Une *ronde* correspond au temps nécessaire pour que tous les processeurs du système qui le peuvent aient fait au moins un pas atomique.

Une mesure temporelle importante dans le cadre de notre étude est le temps de stabilisation. Formellement, on définit deux temps de stabilisation : le temps de stabilisation et le temps de confinement. Le premier est le temps nécessaire au système pour revenir dans une configuration à partir de laquelle son comportement est forcément correct.

Définition 2.29 (Temps de stabilisation) *Le temps de stabilisation d'un système auto-stabilisant \mathcal{S} est le plus grand nombre de rondes nécessaires pour faire passer le système de sa configuration illégitime initiale à la première configuration légitime.*

Le temps de stabilisation est supérieur ou égal au temps de confinement. Ce dernier, introduit par Ghosh et Gupta (1996), exprime le temps à partir duquel le service fourni par le système est correct et utilisable. Pour cela, ils ont décomposé l'état d'un processeur en deux parties : *primaire* (ou *de sortie*) et *secondaire* (ou *interne*). La partie primaire est constitué des variables de sortie caractérisant la spécification du problème et le service

à rendre à l'utilisateur. Les autres variables constituent la partie secondaire et peuvent servir à un traitement interne tout comme à l'établissement d'un état correct du système.

Exemple 2.14 *Considérons un système constitué de n processeurs. A chaque processeur p_i est associé deux variables : Cpt_i un compteur sur le nombre de fichiers imprimés, et $Priv_i$ qui peut avoir pour valeur vrai, si p_i est en train d'exécuter sa section critique et faux dans le cas contraire. Dès lors, la tâche consistant à maintenir l'exclusion mutuelle est satisfaite à partir du moment où, à chaque instant, au plus une variable $Priv$ est à vrai. En d'autres termes, la variable $Priv_i$ constitue la partie primaire de l'état du processeur p_i et Cpt_i la partie secondaire.*

Supposons qu'une panne singulière frappe un système \mathcal{S} . L'état du système suite à cette panne est appelé état à 1-faute et le processeur atteint par cette faute est appelé *processeur fautif*. C'est un processeur dont les valeurs des variables constituant la partie primaire (éventuellement celles de la partie secondaire) de son état ne sont pas correctes. Autrement dit, les valeurs des variables primaires d'un processeur fautif sont différentes des valeurs de ces mêmes variables dans l'une quelconques des configurations légitimes. Aussi, on parlera de *processeur correct sur la sortie* lorsque les valeurs de ses variables constituant la partie primaire sont correctes. De même, un processeur est dit *correct* si les deux parties (primaire et secondaire) de son état sont correctes. Par abus de langage, on parle de partie primaire (respectivement secondaire) correcte lorsque toutes les valeurs des variables la constituant sont correctes. Inversement, on parle de partie primaire (respectivement secondaire) incorrecte lorsque l'une des valeurs des variables la constituant est incorrecte. La notion de configuration correcte sur la sortie découle de la notion de processeur correct sur la sortie :

Définition 2.30 (Configuration correcte sur la sortie) *Soit \mathcal{L} un ensemble de configurations légitimes pour le système auto-stabilisant \mathcal{S} . On appelle configuration correcte (ou légitime) sur la sortie toute configuration L_0 telle que la projection sur les variables de sortie (ou les parties primaires) de L_0 soit égale à la projection sur les variables de sortie d'une configuration de \mathcal{L} .*

A partir d'un état à 1-faute, le temps nécessaire pour que le système soit de nouveau dans une configuration correcte sur la sortie est le temps de confinement ou de stabilisation sur la sortie.

Définition 2.31 (Temps de confinement) *Le temps de confinement (ou de stabilisation sur la sortie) d'un système auto-stabilisant \mathcal{S} est le plus grand nombre de rondes nécessaires pour faire passer le système de sa configuration initiale à 1-faute à la première configuration légitime sur la sortie.*

Le degré de perturbation est une mesure de la quantité de processeurs qui changeront les valeurs de leurs variables primaires pour que le système passe d'une configuration à 1-faute à une configuration correcte sur la sortie.

Définition 2.32 (Degré de perturbation) *Soit \mathcal{S} un système auto-stabilisant pour l'ensemble des configurations légitimes \mathcal{L} . Le degré de perturbation de \mathcal{S} est le plus grand nombre de processeurs qui changeront leurs parties primaires à partir d'une configuration à 1-faute jusqu'à une configuration correcte sur la sortie.*

A partir d'une configuration correcte sur la sortie, les parties primaires de tous les processeurs du système sont correctes. A partir d'un tel état, les processeurs ne vont plus changer les valeurs de leurs variables de sortie. Autrement dit, à partir d'un état correct sur la sortie, seules les variables des parties secondaires des processeurs peuvent changer et ceci pour amener le système dans une configuration légitime.

Définition 2.33 (Protection sur la sortie) *Un système \mathcal{S} auto-stabilisant pour un ensemble de configurations légitimes \mathcal{L} respecte la propriété de protection sur la sortie si, quelle que soit la configuration correcte sur la sortie I , et quelle que soit la configuration légitime L atteinte à partir de I , tout processeur correct sur la sortie dans I est correct sur la sortie à partir de la configuration I et jusqu'à la configuration L .*

2.7 Conclusion

Dans ce chapitre, nous avons passé en revue des notions nécessaires pour la modélisation des systèmes répartis sous forme de graphes. Un système réparti est représenté par ses composants de base, les processeurs et les liens de communication, tous deux modélisés par des systèmes de transitions.

Pour analyser les différents comportements d'un système, nous avons introduit la notion d'exécution ainsi que différentes propriétés qu'une exécution peut vérifier. Nous avons en particulier mis l'accent sur les propriétés d'équité, qui portent sur l'ordre d'apparition des actions dans les exécutions. A ce titre, nous avons introduit la notion de démon, qui est un mécanisme extérieur permettant de caractériser l'ordre dans lequel les actions vont être effectuées dans une exécution.

Nous avons proposé une modélisation classique des algorithmes répartis par un modèle à états. Ensuite, nous avons défini la notion d'auto-stabilisation selon les trois propriétés suivantes :

- la convergence vers un ensemble de configurations légitimes,
- la correction des exécutions à partir de toute configuration légitimes,
- la clôture de l'ensemble des configurations légitimes.

Nous avons alors exposé diverses méthodes classiques utilisées pour prouver ces trois propriétés d'un système auto-stabilisant. Enfin, nous avons présenté différentes mesures de complexité utilisées dans les systèmes répartis pour comparer les performances des algorithmes auto-stabilisants. Nous avons distingués deux types de complexité : la complexité

en espace et la complexité en temps. Concernant cette dernière, nous avons introduit les deux mesures classiques suivantes :

- le temps de stabilisation, qui est le temps nécessaire au système pour revenir dans une configuration à partir de laquelle son comportement est forcément correct ;
- le temps de confinement, qui est le temps nécessaire à un système pour passer d'un état atteint par une faute singulière à un état légitime sur la sortie.

Chapitre 3

Coopération auto-stabilisante¹

Résumé. *Dans ce chapitre, nous nous intéressons aux algorithmes de synchronisation entre deux robots voisins afin d'établir une communication temporaire point-à-point via des interfaces sans fil. Nous présentons brièvement un algorithme d'ordonnement des déplacements dans un système de robots dont la spécificité est que les liens ont une durée de vie limitée, correspondant au temps pendant lequel les robots sont à portée les uns des autres. Après avoir exposé les principes fondamentaux de l'algorithme et les hypothèses nécessaires à son bon fonctionnement, nous utilisons les réseaux de Petri comme outil pour prouver sa correction. Ensuite, nous décrivons formellement sa transformation en un algorithme auto-stabilisant. Le comportement d'un robot est alors événementiel : l'occurrence d'un événement déclenche l'exécution d'un code dépendant de l'état courant du robot. Le comportement de ce nouvel algorithme se modélise par une chaîne de Markov à temps discret. Les résultats théoriques de ce domaine, tels que les conditions d'agrégation forte, permettront de réduire cette chaîne en une chaîne de Markov finie, ce qui servira ensuite pour établir la preuve de stabilisation de l'algorithme. Pour cela, nous définissons une condition suffisante de stabilité sur les états du système et nous montrons qu'à partir d'une classe d'états initiaux quelconques, il existe un chemin allant de cette classe vers la classe d'états stables dans la chaîne de Markov finie précédemment obtenue.*

¹Ce chapitre s'appuie en partie sur les résultats présentés dans El Haddad et Haddad (2003b,a).

3.1 Introduction

“Dans les usines, sur le champ de bataille, sur Mars ou au sein des foyers, les robots sont une réalité du monde d’aujourd’hui. Mais l’ère du robot solitaire est sans doute révolue. Comme les autres machines qui nous entourent, les robots, devenus autonomes et mobiles, apprennent à se connecter, à communiquer et à se synchroniser entre eux.”

Fiévet, 2004

L’une des principales motivations du développement de la robotique mobile demeure la substitution à l’être humain en milieu hostile. Les robots mobiles autonomes, qui réalisent des tâches sans intervention d’un opérateur, sont nécessaires dans plusieurs domaines d’application (e.g., militaire, exploration spatiale ou sous-marine . . .), et peuvent accomplir diverses tâches dans des environnements dynamiques. De tels systèmes sont de plus en plus importants pour plusieurs raisons, parmi lesquelles la minimisation des risques pour l’être humain et la réduction des coûts des opérations.

Pour certaines tâches, telles que l’exploration spatiale, il est plus sûr et même nécessaire d’utiliser non pas un mais plusieurs robots qui coopèrent et se coordonnent pour réaliser la tâche en question. Par exemple, l’un des objectifs de la mission MER² était de faire atterrir deux robots d’exploration MER-A (*Spirit*) et MER-B (*Opportunity*) à la surface de Mars, à la recherche de preuves de la présence d’eau. Depuis Janvier 2004, *Spirit* et *Opportunity* se déplacent sur plusieurs kilomètres à la surface de la planète rouge, explorant chacun une zone géologique qui lui est affectée pour la mission. Chacun de ces robots navigue de manière autonome en ramassant des échantillons dans le but de les analyser ou de les rapatrier. Les robots agissent aussi collectivement en communiquant leurs données et en unissant leurs ressources pour prendre des mesures sismiques, météorologiques ou astronomiques. Ainsi, parmi les tendances actuelles en robotique (Sellem et Luzeaux, 2000; Luzeaux, 2000; Defago et Konagaya, 2002), il y a un intérêt grandissant vers la mobilité, la coopération en environnement dynamique et la tolérance aux pannes.

Bref état de l’art. L’idée d’un nombre important de robots autonomes, communiquant entre eux ou avec un point central, n’est pourtant pas nouvelle. Les prémisses du concept ont été imaginés par Brooks et Flynn (1989), arguant que *“le caractère par essence redondant d’un nombre important de machines autonomes et indépendantes augmente la probabilité d’acquisition d’un grand nombre de données et, plus généralement, du succès de la mission”*. Les auteurs défendent la thèse qu’il est largement préférable de conquérir l’espace et de découvrir des planètes inconnues par le biais de milliers, voire de millions, de petits robots autonomes, plutôt qu’en envoyant un seul robot d’une taille imposante.

²Mars Exploration Rovers, mission exploratoire de la planète MARS.

Ils comparent les travaux qui pourraient être menés par de tels robots à ceux d'une colonie de fourmis, les robots pouvant signaler à tout moment leur position et leur désir de communiquer les données recueillies. Le concept a fait son chemin et de multiples travaux de recherche sont désormais consacrés aux robots *Swarm* (qui signifie *fourmiller* en anglais) désignant des groupes composés d'un grand nombre de machines autonomes, mais qui sont capables de communiquer entre elles et d'adopter des comportements collectifs (e.g. Drogoul et Fresneau, 1998). Parmi les divers modèles étudiés dans la littérature, certains privilégient les interactions individuelles entre congénères (e.g. Parker, 1998). Dans la mesure où les possibilités d'interprétation des mouvements des autres robots dépassent en général la compétence des individus, la communication devient alors capitale. Nous nous intéressons ici aux algorithmes de synchronisation entre deux robots voisins afin d'établir une communication temporaire point-à-point via des interfaces sans fil. De nombreux protocoles de communications ont été conçus pour de tels systèmes : Hu et al. (1998) et Prencipe (2001) supposent que les robots et liens de communications sont fiables, alors que Hatzis et al. (1999) et Bracka et al. (2002) supposent que les liens ont une durée de vie limitée (i.e., le temps pendant lequel les robots sont à portée les uns des autres). Pour plus de détails sur les différents travaux en robotique mobile collective, le lecteur pourra se reporter à l'article de synthèse de Cao, Fukunaga et Kahng (1997).

Aucun des travaux mentionnés précédemment ne prend en compte la détection ni la correction des défaillances causées par l'interaction des robots (e.g., l'imprécision des capteurs), ou encore les défaillances liées à l'environnement méconnu dans lequel ils opèrent. En effet, de nos jours, le mythe de la machine capable d'affronter n'importe quel type de problème cède le pas à la réalisation de systèmes tolérants aux pannes temporaires. Dans cet esprit, de nombreuses approches furent développées afin d'obtenir des systèmes robotisés de plus en plus performants. Parmi celles-ci, Suzuki et Yamashita (1999) ont proposé un algorithme d'ordonnancement des déplacements des robots en formation. L'objectif de ce comportement est de pouvoir permettre à une équipe de robots de réaliser des missions d'exploration ou de reconnaissance de manière coopérative. Le déplacement en formation est important dans ce genre de mission car il permet à chaque membre du groupe de ne se focaliser que sur une partie précise de l'environnement tout en bénéficiant éventuellement des observations des autres robots. Suzuki et Yamashita ont présenté un algorithme auto-stabilisant de synchronisation des déplacements des robots pour atteindre un point fixe donné en un temps fini. Afin de maintenir une formation précise et cohérente tout au long des déplacements du groupe, chaque robot calcule la position qu'il doit rejoindre en fonction de sa position courante et de celles des autres robots actifs du groupe. Ainsi, sous l'hypothèse que les robots gardent l'historique de leurs déplacements, on maintient une distribution stable au sein d'une formation donnée. Flocchini et al. (1999) ont également

proposé un algorithme auto-stabilisant pour le déplacement coopératif en formation, mais dans le cadre d'un modèle plus simple. En effet, contrairement au modèle utilisé par Suzuki et Yamashita, les robots ne gardent pas l'historique de leurs déplacements, ne dispose pas du sens de l'orientation et ne peuvent se déplacer, en une étape, que d'une distance perceptible par les autres robots.

Plus récemment, Zhang et Arora (2002) ont proposé un algorithme auto-stabilisant permettant aux nœuds d'un réseau sans fil de s'organiser en *cellules* de rayon R . Une cellule est formée d'un nœud distingué, appelé tête de la cellule, et de nœuds associés qui ne communiquent ensemble que via la tête de la cellule. Étant donné un rayon R , l'algorithme vise à organiser les nœuds en des cellules de rayon approximativement R , chaque nœud appartenant à une et une seule cellule et étant relié par un chemin à la tête de la cellule. L'auto-stabilisation maintient la stabilité de la structure en cellules dans un système à grande échelle en garantissant : d'une part la réorganisation des nœuds en cellules en cas de suppression ou d'ajout de nouveaux nœuds, d'autre part la seule participation des nœuds dans un rayon restreint en cas de déplacement d'une tête de cellule. Ainsi, cet algorithme permet un passage à l'échelle en respectant la connaissance d'un nombre constant d'identités de nœuds voisins et le confinement de fautes dans une région restreinte autour du nœud fautif.

Par ailleurs, Demirbas, Arora et Gouda (2003) ont proposé un algorithme auto-stabilisant pour le problème "Poursuite-Evasion" dans un réseau de capteur. Dans ce problème, le poursuivant et l'intrus sont deux robots mobiles qui se déplacent sur les nœuds d'un réseau, avec l'hypothèse que le poursuivant a une vitesse supérieure à celle de l'intrus. Néanmoins, la stratégie de l'intrus est inconnue alors que celle du poursuivant dépend de l'état courant du nœud sur lequel il se trouve. L'algorithme consiste à ce que les nœuds du système maintiennent un arbre enraciné sur le nœud où l'intrus se trouve. Lorsque l'intrus change de nœud, l'arbre est reconstruit dynamiquement de façon à ce qu'il soit toujours enraciné sur le nœud où réside l'intrus.

Notre contribution se situe dans la lignée de ces différents algorithmes de communications auto-stabilisants. L'algorithme que nous proposons se différencie de ces derniers par la formalisation que nous donnons au modèle de passage de messages de robots mobiles, et du fait que restons sous l'hypothèse que la communication se fait par voie directe entre deux robots.

3.2 Ordonnancement des déplacements des robots

Dans le contexte des systèmes de robots mobiles, Bracka, Midonnet et Roussel (2002) ont proposé un algorithme d'ordonnancement basé sur une stratégie de déplacement des

robots assurant une communication entre robots distants en un temps borné. Pour assurer cette condition, les auteurs restreignent le modèle en posant des contraintes sur la topologie du réseau et les déplacements des robots.

3.2.1 Les hypothèses du modèle

Si aucune contrainte n'est posée sur les déplacements, on ne peut pas assurer que la communication aura lieu en temps borné. En effet, il est toujours possible de trouver un déplacement dans lequel les robots ne se rencontrent jamais. Avant de préciser ces contraintes, nous commençons par la description des composants du réseau.

- Le système est composé d'un ensemble de m robots anonymes (les identités ne seront donc pas utilisées dans l'algorithme), notés (r_1, \dots, r_m) et de N points de rendez-vous.
- A chaque point de rendez-vous (ou *location*) est associé un unique identifiant numérique. Nous notons (ℓ_1, \dots, ℓ_N) l'ensemble ordonné des points de rendez-vous du système.
- Un point de rendez-vous est associé à une paire de robots et la communication ne peut avoir lieu que si les deux robots concernés sont présents au point de rendez-vous. La localisation de ce dernier est fixe et connue par les deux robots.
- Chaque robot r_i maintient, dans sa mémoire incorruptible, un tableau MP_i trié par ordre croissant des identificateurs de ses points de rendez-vous. La taille de ce tableau est n_i , et un élément de ce tableau, noté $MP_i[j]$, contient l'identificateur donné par la fonction $f(i, j)$ pour $1 \leq i \leq m$ et $0 \leq j \leq n_i - 1$. Cet identificateur est associé au $j^{\text{ième}}$ point de rendez-vous de r_i .

Les hypothèses prises sur la topologie du réseau et les déplacements des robots sont les suivantes :

- Le réseau est "connexe" : entre deux robots r_i et $r_{i'}$, il existe une séquence de robots $r_i = r_{i_0}, r_{i_1}, \dots, r_{i_K} = r_{i'}$ tel que pour tout $0 \leq k < K$, r_{i_k} et $r_{i_{k+1}}$ ont un point de rendez-vous en commun.
- Les robots s'attendent mutuellement à leur point de rendez-vous. En d'autres termes, le premier robot qui arrive sur le point de rendez-vous attend son homologue avant de continuer son déplacement.

Un exemple de système composé de 4 robots et de 4 points de rendez-vous, illustrant les différentes hypothèses précédentes, est présenté sur la figure 3.1. Malgré ces hypothèses sur les déplacements des robots, il est toujours possible de trouver un ordonnancement dans lequel chacun des robots reste bloqué sur un point de rendez-vous et ne

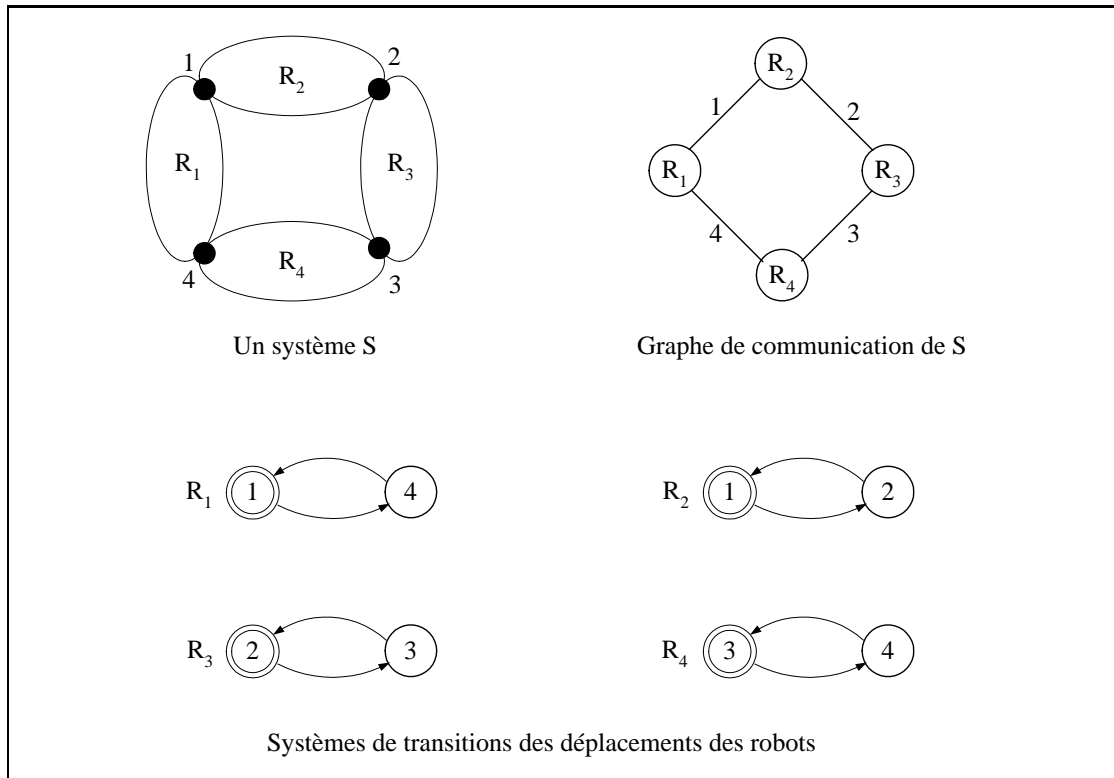


FIG. 3.1 – Un exemple de système composé de 4 robots et sa modélisation

se déplace jamais. C'est pourquoi il est nécessaire d'établir un ordonnancement préalable des déplacements des robots.

3.2.2 Algorithme d'ordonnancement non auto-stabilisant

Afin d'établir un ordonnancement, une numérotation arbitraire des points de rendez-vous est choisie et chaque robot effectue un parcours itératif de ses points de rendez-vous. Le graphe de communication correspondant est $G = (V, E)$, où l'ensemble des sommets V correspond à l'ensemble des robots et l'ensemble E aux points de rendez-vous. Il existe un lien $\{i, j\}$ dans E si et seulement si les robots r_i et r_j ont un point de rendez-vous en commun. Ce lien est valué par l'identificateur du point de rendez-vous (voir figure 3.1).

Étant donné un graphe G et une numérotation sur E , l'algorithme d'ordonnancement construit, pour chaque robot r_k , un système local de transition représentant le parcours de ses points de rendez-vous (voir figure 3.1). Chaque système de transition \mathcal{S}_k est défini par :

- l'ensemble des états \mathcal{Q}_k représentant les points de rendez-vous. Il existe un état q_i pour chaque arc valué q_i du robot r_k . L'état initial de l'automate est $\min(\mathcal{Q}_k)$;

- l'ensemble de transitions T_k représentant les déplacements entre points de rendez-vous. Il existe une transition de q_i à q_j si et seulement si :
 - $q_i < q_j$ et il n'existe pas d'état $q \in Q_k$ tel que $q_i < q < q_j$, ou
 - $q_i = \max(Q_k)$ et $q_j = \min(Q_k)$.

L'idée de l'algorithme de Bracka, Midonnet et Roussel (2002) est que chaque robot r_k parcourt ses n_k points de rendez-vous dans l'ordre croissant de leurs numéros. Il s'agit de partir du point de rendez-vous ayant le plus petit numéro, puis de visiter tous les autres points de manière unidirectionnelle et répétitive. Ainsi, chaque point de rendez-vous est visité une infinité de fois.

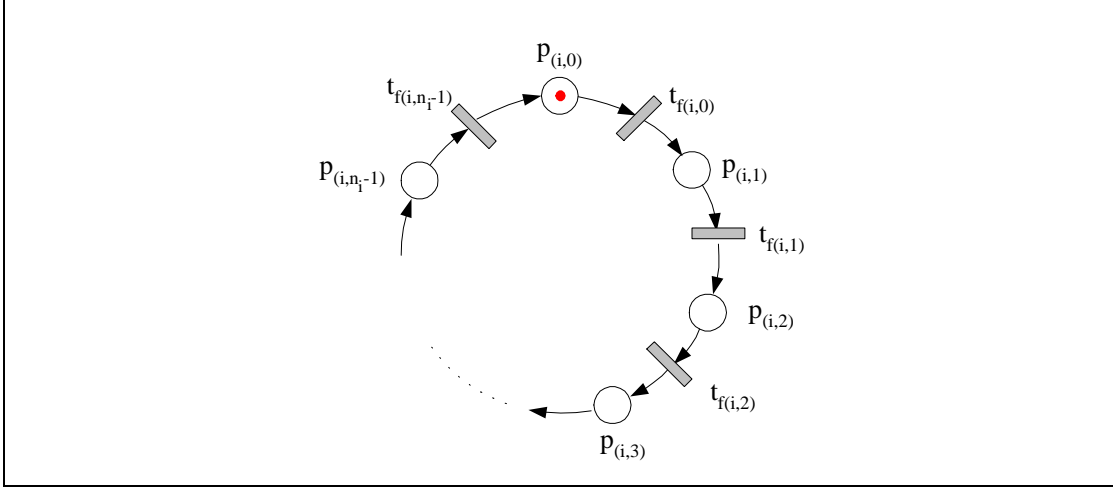
Pour démontrer que l'ordonnement produit est exempt de blocage (partiel et total), Bracka et al. construisent, à partir des systèmes de transitions locaux, le système de transition produit représentant l'état global du réseau à tout instant. Ce système est utilisé pour que l'ordonnement soit exempt de blocage partiel ou total. Tout d'abord, les auteurs ont prouvé que quelle que soit la numérotation choisie pour les points de rendez-vous, le système n'est jamais bloqué. En effet, il y a toujours un robot qui se déplace dans le système, puisque tout état du système de transition global, atteint à partir de l'état initial, a un successeur. Pour prouver l'absence de blocage partiel, ils prouvent que quelle que soit la suite de transitions (i.e., de déplacements) dans le système de transition produit, tous les systèmes de transitions locaux changent d'état au bout d'un nombre fini de transitions du système produit.

Notre contribution consiste à utiliser le modèle des réseaux de Petri (RdP) comme support d'une preuve plus courte et plus simple de correction de cet algorithme. Pour une bonne introduction aux réseaux de Petri, le lecteur pourra se reporter par exemple à des ouvrages comme Reisig (1985) et Diaz (2001).

3.3 Preuve de correction

Dans cette section, nous présentons une preuve qui sera à la base de la version stabilisante de l'algorithme. Pour chaque robot r_i , un réseau de Petri modélise le parcours local de ses points de rendez-vous. Le produit synchronisé des réseaux locaux est un réseau de Petri global modélisant l'état global du système à tout instant. Le réseau de Petri de la figure 3.2 représente le comportement d'un robot r_i , et se définit par le tuple $N_i = (P_i, T_i, Pré_i, Post_i, M0_i)$ où :

- $P_i = \{p_{(i,0)}, \dots, p_{(i,j)}, \dots, p_{(i,n_i-1)}\}$ est un ensemble fini et non vide de places représentant l'ensemble des points de rendez-vous. La présence d'un jeton à la place $p_{(i,j)}$ signifie que r_i est soit en déplacement vers son $j^{\text{ième}}$ point de rendez-vous, soit présent sur ce dernier mais *en attente* de son homologue.

FIG. 3.2 – Le cycle des visites du robot r_i

- $T_i = \{t_{f(i,0)}, \dots, t_{f(i,j)}, \dots, t_{f(i,n_i-1)}\}$ est un ensemble fini et non vide de transitions. La condition de franchissement d'une transition $t_{f(i,j)}$ est que les deux robots, r_i et son homologue, soient présents sur le $j^{\text{ième}}$ point de rendez-vous de r_i . Le franchissement de $t_{f(i,j)}$ représente le déplacement des deux robots, chacun vers son point de rendez-vous suivant.
- $Pré_i$ est la matrice d'incidence arrière, définie de $P_i \times T_i$ dans $\{0, 1\}$ comme suit :

$$Pré_i(p, t) = \begin{cases} 1 & \text{si } p = p_{(i,j)} \text{ et } t = t_{f(i,j)}, \\ 0 & \text{sinon.} \end{cases}$$

- $Post_i$ est la matrice d'incidence avant, définie de $P_i \times T_i$ dans $\{0, 1\}$ comme suit :

$$Post_i(p, t) = \begin{cases} 1 & \text{si } p = p_{(i,(j+1) \bmod n_i)} \text{ et } t = t_{f(i,j)}, \\ 0 & \text{sinon.} \end{cases}$$

- Le marquage initial, noté $M0_i$, définit le nombre de jetons contenus dans toutes les places du réseau. $M0_i(p_{(i,j)})$ définit le marquage initial de la place $p_{(i,j)}$ comme suit :

$$M0_i(p_{(i,j)}) = \begin{cases} 1 & \text{si } j = 0, \\ 0 & \text{sinon.} \end{cases}$$

Le modèle global du système est un réseau de Petri défini par le tuple $N = (P, T, Pré, Post, M0)$ où :

- $P = \bigsqcup P_i$ est l'union des toutes les places des réseaux de Petri locaux ;

- $T = \bigcup T_i$, est l'union (non disjointe) de toutes les transitions des réseaux de Petri locaux ;
- $Pré$ et $Post$ sont les matrices d'incidence arrière et avant, définies de $P \times T$ dans $\{0, 1\}$ comme suit :

$$Pré(p, t) = \begin{cases} Pré_i(p, t) & \text{si } p \in P_i \text{ et } t \in T_i, \\ 0 & \text{sinon.} \end{cases}$$

$$Post(p, t) = \begin{cases} Post_i(p, t) & \text{si } p \in P_i \text{ et } t \in T_i, \\ 0 & \text{sinon.} \end{cases}$$

- $M0$ est le marquage initial défini par $M0(p) = M0_i(p)$ si $p \in P_i$.

Exemple 3.1 *Considérons un système constitué de 6 robots et 6 points de rendez-vous. La figure 3.3 représente le réseau de Petri global correspondant à ce système. En ordonnant les points de rendez-vous de chacun des robots, nous obtenons le tableau suivant :*

| Robots | r_1 | r_2 | r_3 | r_4 | r_5 | r_6 |
|-----------------------|-------|-------|-------|-------|-------|-------|
| Points de rendez-vous | 1 | 2 | 1 | 3 | 5 | 6 |
| | 3 | 4 | 2 | 4 | | |
| | 5 | 6 | | | | |

Afin de modéliser la synchronisation entre deux robots par rendez-vous, nous utilisons la technique de fusion des transitions. Remarquons que le graphe global obtenu appartient à la famille des *graphes d'événements* dans lesquels les transitions ne sont jamais en conflit, puisqu'une place est entrée (et sortie) d'une seule transition. Dans les réseaux de Petri, l'absence de blocage correspond à la vivacité, c'est-à-dire à l'existence d'une séquence de franchissements à partir du marquage initial comportant exactement une occurrence de toutes les transitions. Autrement dit, quel que soit l'état du système, toute transition est potentiellement franchissable dans le futur. Pour cette famille de graphes, de nombreuses propriétés comportementales sont caractérisées par des conditions structurelles. Nous nous limiterons ici à un seul lemme. Nous rappelons sa preuve, puisqu'elle contient des éléments que nous utiliserons ultérieurement pour la preuve de l'auto-stabilisation.

Lemme 3.1 (VIVACITÉ D'UN GRAPHE D'ÉVÉNEMENTS). – *Si N est un graphe d'événements tel que tout circuit élémentaire est initialement marqué, alors N est vivant.*

Preuve. Remarquons, tout d'abord, que le nombre de jetons d'un circuit élémentaire est invariant dans un graphe d'événements, puisqu'il ne peut y avoir de transition entrante ni sortante.

Supposons que tous les circuits soient initialement marqués, alors pour tout marquage M accessible, ils le sont aussi, d'après la remarque précédente. Pour ce marquage M , nous définissons deux relations binaires :

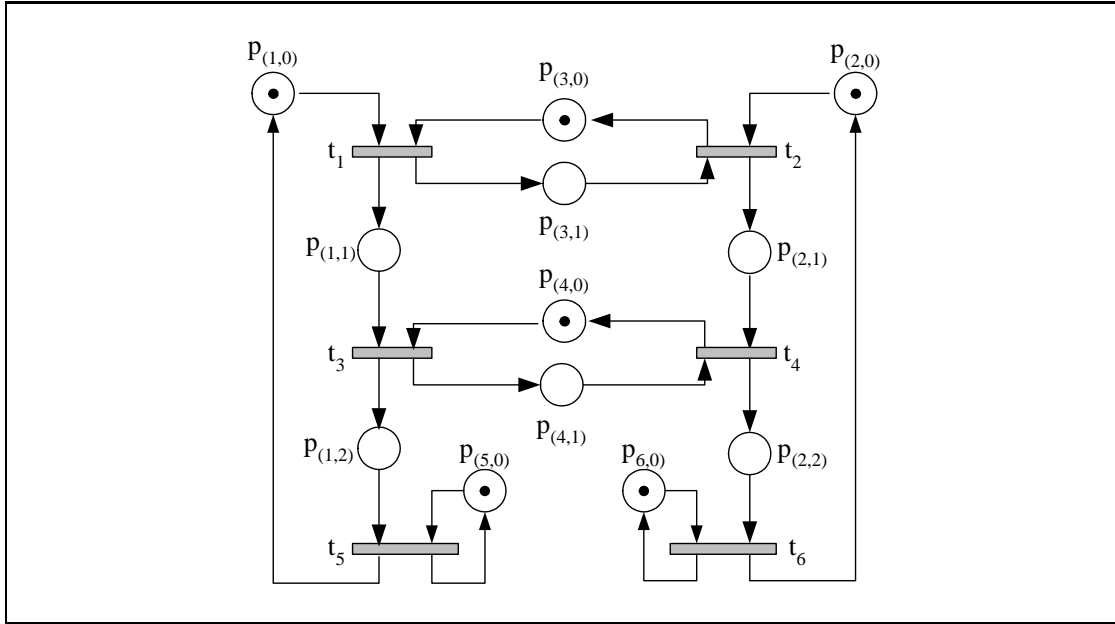


FIG. 3.3 – Le réseau de Petri global d’une instance de l’algorithme

- $t \text{ aide}_M t'$, si et seulement si il existe une place non marquée de sortie de t et d’entrée de t' ;
- $t \text{ précède}_M t'$, la fermeture réflexive et transitive de aide_M .

Montrons que précède_M est une relation d’ordre partiel. Si tel n’est pas le cas, nous avons deux transitions t et t' telles que $t \text{ précède}_M t'$ et $t' \text{ précède}_M t$. D’après la définition de précède_M , cela signifie qu’il existe un chemin de t à t' et un chemin de t' à t où aucune place n’est marquée. En les composant, on obtient un circuit dont on peut extraire un circuit élémentaire non marqué, ce qui est impossible.

Toute relation d’ordre partiel sur un ensemble fini peut s’étendre en (au moins) une relation d’ordre total. Soit t_1, \dots, t_n la liste ordonnée des transitions en cet ordre total. Nous affirmons que $t_1 \dots t_n$ est une séquence de franchissements à partir de M . En effet, t_1 est franchissable, puisque toutes ses places entrées sont marquées. Par induction, le franchissement de la séquence $t_1 \dots t_i$ conduit à un nouveau marquage M' . Toutes les places entrées de t_{i+1} sont marquées dans M' soit parce qu’elles le sont dans M , soit comme conséquence du franchissement d’une certaine transition t_j tel que $j \leq i$. Rappelons que toute place entrée de t_j n’est entrée d’aucune autre transition. Donc la séquence de franchissement peut être étendue jusqu’à t_{i+1} . Par conséquent, le réseau est vivant. ■

La proposition suivante établit la correction de l’algorithme. Sa preuve découle presque directement du lemme précédent.

Proposition 3.2 *Soit N le graphe d'événements représentant l'algorithme pour un réseau donné, alors N est vivant.*

Preuve. Prenons un circuit \mathcal{C} de N . Notons t_k la transition de plus petit identificateur dans \mathcal{C} , $p_{(i,j)}$ la place d'entrée de t_k , et $t_{k'}$ la transition d'entrée de $p_{(i,j)}$ dans \mathcal{C} . Par construction de N , $k = f(i, j)$ et $k' = f(i, (j - 1) \bmod n_i)$. Ce choix de t_k implique que $k < k'$. Puisque f est croissante vis-à-vis de son second paramètre, l'unique valeur possible pour j est 0. Ainsi, $p_{(i,j)}$ est la place $p_{(i,0)}$. Puisque $p_{(i,0)}$ est initialement marquée, nous avons prouvé que tout circuit de N contient une place initialement marquée. D'après le lemme 3.1 précédent, N est donc vivant. ■

Ce résultat se généralise au cas de rendez-vous n-aires. Ce schéma de rendez-vous mettant en jeu plusieurs robots diminue la flexibilité du système. Aussi, nous nous restreindrons au schéma de rendez-vous binaire.

3.4 Ordonnement auto-stabilisant

3.4.1 Les hypothèses du modèle

Dans cette section, nous proposons de doter l'algorithme précédent d'un mécanisme auto-stabilisant. Nous procédons, tout d'abord, à la description des hypothèses additionnelles. Nous décrivons les variables locales utilisées par chaque robot.

- Chaque robot r_i dispose d'un compteur, contenant une valeur de suspension.
- Le déplacement d'un robot d'un point de rendez-vous vers un autre coûte au plus 1 unité de temps (*u.t.*). Cette hypothèse peut être facilement satisfaite avec un choix judicieux de l'unité de temps.
- Chaque robot possède un capteur lui permettant de connaître sa position courante dans le réseau.

Chaque robot r_i maintient les variables suivantes :

- $timeout_i$: une variable contenant la valeur d'un délai de suspension, à valeur réelle dans l'intervalle $[0 \dots N]$;
- $MP_i[0 \dots n_i - 1]$: un tableau contenant les identifiants de ses points de rendez-vous triés par ordre croissant ;
- $position_i$: une variable contenant la valeur du capteur du robot. Elle représente la position courante du robot r_i , à valeur dans $\{0, \dots, n_i - 1\} \cup nowhere$. Elle prend soit la valeur *nowhere* lorsque r_i est en déplacement d'un point de rendez-vous à un autre, soit la valeur de l'identifiant du point de rendez-vous sur lequel il se trouve ;

- $statut_i$: une variable représentant l'état du robot. Elle prend la valeur *moving* si le robot r_i est en déplacement d'un point de rendez-vous à un autre, et la valeur *waiting* s'il est en attente sur un point de rendez-vous.

3.4.2 Algorithme d'ordonnement auto-stabilisant

Nous proposons ici une variante de l'algorithme précédent dotée d'un aspect auto-stabilisant. Le comportement d'un robot est événementiel : l'occurrence d'un événement déclenche l'exécution d'un code dépendant de l'état courant du robot. Dans notre cas, deux événements sont possibles : l'*expiration du timeout* et la *détection de l'homologue*. Ce dernier exige la présence des deux robots concernés au point de rendez-vous. Nous supposons que l'arrivée d'un robot à un point de rendez-vous ne constitue pas un événement. Puisqu'un robot réarme son timeout de $1u.t.$ avant de se diriger vers un nouveau point de rendez-vous, son timeout expirera après son arrivée à ce point, ce qui lui permet d'exécuter les actions correspondantes à son arrivée. Le point critique dans ce mécanisme est l'hypothèse que le déplacement entre deux points de rendez-vous dure exactement $1u.t.$. L'activité d'un robot r_i vis à vis des événements est gérée par une variable $statut_i$ dont la valeur est soit *moving*, soit *waiting*. Quand un robot est actif, c'est-à-dire son statut est *moving*, il ne peut ni détecter son homologue, ni être détecté par ce dernier. Autrement dit, l'arrivée d'un robot r_i sur un point de rendez-vous, même si son homologue y est déjà, ne déclenche pas automatiquement l'événement *détection de l'homologue*. La phase d'échange de données n'est engagée qu'après expiration du timeout de r_i .

Une description formelle de l'algorithme est donnée ci-après (Algorithme 1). Le robot courant est r_i . L'algorithme est décrit suivant les quatre activités d'un robot : SYNC, WAIT, RECOVER et MISS. Les actions SYNC et WAIT correspondent aux actions du protocole originel (cf. section 3.2). Afin de savoir si l'expiration du timeout d'un robot correspond à son arrivée à un point de rendez-vous, nous utilisons la variable $statut_i$. En effet, cette dernière prend la valeur *moving* quand le robot se dirige vers un nouveau point de rendez-vous, et la valeur *waiting* quand son timeout expire sur un point de rendez-vous. Cependant, l'action WAIT est différente de celle de l'algorithme originel, puisque r_i arme son timeout de $Nu.t.$ (cf. la sous-section 3.5.2). Si l'expiration du timeout d'un robot r_i correspond à son arrivée à un point de rendez-vous, il déclenche l'exécution de l'action WAIT, même si son homologue y est déjà. Cette action changera le statut de r_i à *waiting* permettant ainsi, à r_i et à son homologue, d'exécuter simultanément leurs actions SYNC respectives.

L'occurrence d'une panne déclenche l'exécution de l'action RECOVER, qui est la première action exécutée par le robot fautif. Ce cas de figure n'est possible que si le robot est en déplacement d'un point de rendez-vous vers un autre au moment de la panne. Il

arme alors son timeout de $1u.t.$ et se dirige vers son premier point de rendez-vous.

L'action qui permettra la stabilisation de l'algorithme est MISS. Elle est exécutée soit initialement, soit à l'expiration du timeout d'un robot en train d'attendre son homologue. Dans ce cas, le robot r_i réarme son timeout de $1u.t.$ et choisit soit de rester sur place, soit de se diriger vers son premier point de rendez-vous. Ce choix est généré aléatoirement selon une distribution de probabilité uniforme, ce qui correspond à la fonction *Uniform-choice* qui affecte à son unique paramètre ($choice_i$) l'une des valeurs de l'ensemble $\{0, 1\}$.

Une exécution de l'algorithme se présente sous la forme d'une séquence infinie $\{t_n, A_n\}_{n \in \mathbb{N}}$ où $\{t_n\}_{n \in \mathbb{N}}$ est une suite strictement croissante et chaque A_n est un ensemble non vide d'actions déclenchées à l'instant t_n (au plus deux actions par robot, dans le cas où il exécute WAIT et tout de suite après SYNC). Dans le cadre de ce formalisme, nous définissons maintenant une *exécution stabilisante*.

Définition 3.1 *Une exécution $\{t_n, A_n\}_{n \in \mathbb{N}}$ de l'algorithme est stabilisante si le nombre d'occurrences des actions RECOVER et MISS est fini.*

En d'autres termes, après un temps fini l'algorithme aura le même comportement que l'algorithme originel. Rappelons que l'action RECOVER n'est exécutée qu'une seule fois au plus par chaque robot. La prochaine section est consacrée à une description informelle de la preuve du résultat suivant, qui sera prouvé formellement par la suite :

Proposition 3.3 *Partant d'un état quelconque, la probabilité qu'une exécution se stabilise est égale à 1.*

3.4.3 Idée de la preuve

Nous nous restreindrons sans perte de généralité à l'état initial obtenu après que chaque robot ait exécuté au moins une action. Ceci nous évite de prendre en compte l'action RECOVER.

Tout d'abord, nous supposons que le temps d'exécution du code est instantané et négligeable par rapport à celui du déplacement des robots. Ainsi, la seule source de non déterminisme dans notre algorithme est le choix uniforme de l'action MISS, puisque tout déplacement d'un point vers un autre prend exactement $1u.t.$. Par conséquent, la sémantique probabiliste de l'algorithme est définie par une chaîne de Markov à temps discret que nous décrivons dans la suite (cf. section 3.5.1). Pour un rappel des concepts de base des chaînes de Markov, le lecteur pourra se reporter à Kemeny et Snell (1960).

L'état d'un robot est défini par son statut, par l'identifiant de son point de rendez-vous courant et par son timeout qui est divisé en une valeur entière et une valeur résiduelle comprise entre 0 et 1. De la sorte, un état de la chaîne de Markov est le produit des états locaux de tous les robots.

Algorithme 1 *Algorithme auto-stabilisant d'ordonnancement des déplacements de r_i*

```

Constantes    $N, n_i \in \mathbb{N}; MP_i[0 \dots n_i - 1] \in \mathbb{N};$ 
Timer        $timeout_i \in [0 \dots N + 1];$ 
Sensor       $position_i \in \{0, \dots, n_i - 1\} \cup \{nowhere\};$ 
Variables    $statut_i \in \{waiting, moving\};$ 
                 $choice_i \in \{0, 1\};$ 

Détection de l'homologue // SYNC
// à l'arrivée de l'un des deux robots concernés,  $r_i$  ou son homologue,
// alors que l'autre l'attend
// Nécessairement  $statut_i$  est waiting
Échange des messages;
Réarme( $timeout_i, 1$ );
 $statut_i = moving$ ;
 $position_i = (position_i + 1) \bmod n_i$ ;
Aller à  $MP_i[position_i]$ ;

Expiration du timeout
si ( $position_i \neq nowhere$ ) et ( $statut_i == moving$ ) alors // WAIT
//  $r_i$  vient d'arriver au point de rendez-vous
  Réarme( $timeout_i, N$ );
   $statut_i = waiting$ ;
finsi

si ( $position_i == nowhere$ ) alors // RECOVER
// recouvrement d'une panne survenue alors que  $r_i$  était en déplacement
// entre deux points de rendez-vous
  Réarme( $timeout_i, 1$ );
   $statut_i = moving$ ;
   $position_i = 0$ ;
  Aller à  $MP_i[position_i]$ ;
finsi

si ( $position_i \neq nowhere$ ) et ( $statut_i == waiting$ ) alors // MISS
// expiration du timeout de  $r_i$  pendant qu'il attend son homologue
  Uniform-Choice( $choice_i$ );
  case ( $choice_i$ )
    0 : Réarme( $timeout_i, 1$ );
    1 : Réarme( $timeout_i, 1$ );
         $statut_i = moving$ ;
         $position_i = 0$ ;
        Aller à  $MP_i[position_i]$ ;
  fincase
finsi

fin

```

Notons que l'ensemble des états est infini puisque le domaine de valeurs résiduelles est un intervalle dans \mathbb{R} . En extrayant les valeurs résiduelles, nous obtenons une relation d'équivalence entre les états basée sur les positions relatives de ces valeurs. Ainsi, nous réduisons notre chaîne à une chaîne de Markov finie respectant les conditions d'agrégation forte.

Ensuite, nous définissons une condition suffisante de stabilité sur les états du système (cf. section 3.5.2). Nous montrons que pour toute exécution partant d'un état satisfaisant cette condition, aucune occurrence de l'action MISS n'est possible. A cette fin, nous remarquons que si un marquage M est sans blocage, alors la relation $aide_M$ introduite dans le lemme 3.1 induit un graphe sans cycle. La longueur d'un chemin étant le nombre d'arcs le constituant, nous définissons $level_M(t)$ comme la longueur du plus long chemin d'extrémité t . La condition suffisante s'exprime alors ainsi :

- un état e est stable si son marquage $M(e)$ est sans blocage, et
- pour tout robot en attente de son homologue (c'est-à-dire, dont le statut est *waiting*), la valeur entière de son timeout est supérieure à $level_M(t)$.

Enfin, nous montrons qu'à partir d'une classe d'états initiaux quelconques, il existe un chemin allant de cette classe vers la classe d'états stables dans la chaîne de Markov agrégée (la relation d'équivalence étant plus fine que la relation de stabilité). Le seul point non trivial est que pour deux états équivalents quelconques s et s' , s est stable si et seulement si s' est stable, puisque la définition de la stabilité ne prend en compte que les valeurs résiduelles égales à 1. Par agrégation, tout chemin dont l'état final est stable correspond, dans la chaîne finie agrégée, à un chemin dont l'état final est l'ensemble des états stables.

Pour trouver ce chemin particulier dans la chaîne de Markov, nous fixons l'issue du choix aléatoire à chaque exécution de l'action MISS : le choix de rester sur place afin de reproduire l'algorithme originel. Supposons que l'état initial, noté e , ne soit pas stable. Nous examinons, selon le marquage de e , les deux cas possibles :

- $M(e)$ est sans blocage : dans ce cas, seules les contraintes temporelles peuvent être construites à l'expiration d'un timeout. Afin de reproduire le comportement de l'algorithme originel, nous imposons aux robots le choix de rester sur place. Nous prouvons qu'une fois la contrainte de temps satisfaite pour un robot, elle le reste par la suite. Ainsi, l'état atteint après que chacun des robots ait exécuté au moins une action SYNC est stable.
- $M(e)$ est avec blocage : dans ce cas, nous simulons le comportement de l'algorithme originel jusqu'au moment où chaque robot se trouve seul sur un point de rendez-vous et a exécuté au moins une fois l'action MISS. A cet instant, tous les timeouts

ont des valeurs inférieures ou égales à 1, et nous imposons à chaque robot de choisir la seconde alternative durant l'exécution de son action MISS. Après la dernière exécution d'une action MISS, chacun des robots est sur le chemin de son premier point de rendez-vous. Dans un tel état, noté e' , le marquage $M(e')$ correspond au marquage initial du réseau N de l'algorithme originel. Par conséquent, le marquage $M(e')$ est sans blocage et la suite du chemin est complétée par le chemin construit dans le cas précédent (cf. section 3.5.3).

3.5 Preuve formelle

Nous nous restreindrons, sans perte de généralité, à l'état initial obtenu après que chaque robot ait exécuté au moins une action. Ceci nous permettra de ne pas prendre en compte l'action RECOVER. Sous cette hypothèse et pour bien comprendre l'algorithme, le graphe d'état d'un robot est présenté sur la figure 3.4.

3.5.1 Sémantique probabiliste

Nous supposons que l'exécution du code est instantanée et négligeable par rapport temps de déplacement des robots. Ainsi, la seule source de non déterminisme dans notre algorithme est le choix uniforme de l'action MISS, puisque tout déplacement d'un point vers un autre prend exactement *1u.t.*. Par conséquent, l'aspect probabiliste de notre algorithme est caractérisé par une chaîne de Markov que nous décrivons dans la suite.

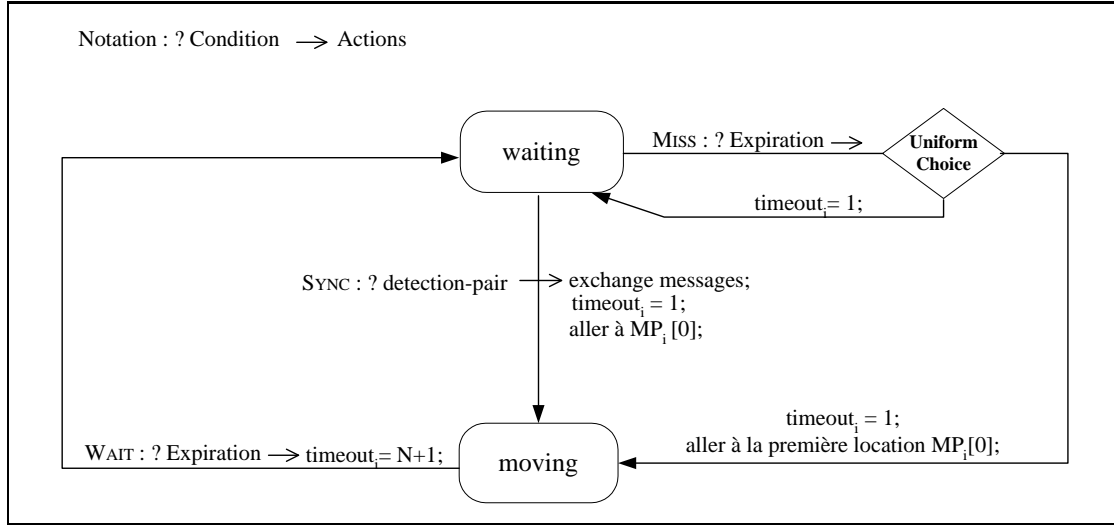
Un état de cette chaîne est composé des spécifications de l'état local de chaque robot du système. L'état d'un robot r_i est défini par un vecteur $\langle s_i, \ell_i, to_i, \alpha_i \rangle$ où :

- s_i est le statut du robot à valeur dans $\{waiting, moving\}$ suivant que r_i attend sur un point ou se dirige vers un point de rendez-vous ;
- ℓ_i est l'identifiant du point de rendez-vous courant ou vers lequel r_i se dirige ;
- to_i est donné par la formule $\lfloor timeout_i \rfloor$ (où $\lfloor x \rfloor$ est plus grand entier inférieur ou égal à x) et prend sa valeur dans $\{0, \dots, N - 1\}$;
- α_i est donné par la formule $\alpha_i = timeout_i - to_i$ et prend sa valeur dans $]0 \dots 1]$.

Nous l'appellerons *valeur résiduelle*.

Puisque nous considérons les états après l'exécution des actions, les valeurs des timeouts ne sont jamais nulles. Ceci explique le choix de l'intervalle de valeurs des α_i . Pour le moment, ces deux dernières variables ne servent qu'à la décompositions des timeouts.

Un état e du système se définit comme le produit des états locaux de tous les robots $e = \prod_{i=1}^m \langle s_i, \ell_i, to_i, \alpha_i \rangle$. Notons que l'ensemble des états est infini et non dénombrable

FIG. 3.4 – Le graphe d'état de r_i

puisque le domaine de α_i est un intervalle dans \mathbb{R} . Cependant, nous pouvons réduire cette chaîne en une chaîne dont l'ensemble d'états est fini grâce à la relation d'équivalence suivante :

Définition 3.2 Deux états $e^1 = \prod_{i=1}^m \langle s_i^1, \ell_i^1, to_i^1, \alpha_i^1 \rangle$ et $e^2 = \prod_{i=1}^m \langle s_i^2, \ell_i^2, to_i^2, \alpha_i^2 \rangle$ sont équivalents si :

1. $\forall i, s_i^1 = s_i^2$;
2. $\forall i, \ell_i^1 = \ell_i^2$;
3. $\forall i, to_i^1 = to_i^2$;
4. $\forall i, j, \alpha_i^1 < \alpha_j^1 \iff \alpha_i^2 < \alpha_j^2$ et $\alpha_i^1 = 1 \iff \alpha_i^2 = 1$

Une classe d'équivalence c de cette relation est définie par : $c = \prod_{i=1}^m \langle s_i, \ell_i, to_i \rangle \times position$ où *position* représente les positions relatives des α_i . Il est évident qu'il existe au plus $m! \cdot 2^{m-1}$ positions possibles, et par conséquent que le nombre de classes d'équivalences est fini. La proposition suivante établit la condition de forte agrégation.

Proposition 3.4 Soient c et c' deux classes d'équivalence, e^1 et e^2 deux états appartenant à la classe c , alors :

$$\sum_{e \in c'} P[e^1, e] = \sum_{e \in c'} P[e^2, e]$$

où P est la matrice de transition de la chaîne de Markov.

Preuve. Considérons deux états équivalents $e^1 = \prod_{i=1}^m \langle s_i, \ell_i, to_i, \alpha_i^1 \rangle$ et $e^2 = \prod_{i=1}^m \langle s_i, \ell_i, to_i, \alpha_i^2 \rangle$. Soit I un sous-ensemble des indices i tel que α_i^1 est la valeur minimale parmi les valeurs résiduelles de e^1 ; dans la suite nous la notons α_{min}^1 . Soit J un sous-ensemble des indices j tel que α_j^2 est la valeur minimale parmi les valeurs résiduelles de

e^2 ; dans la suite nous la notons α_{min}^2 . Puisque e^1 et e^2 sont équivalents, alors $I = J$ et $\forall k, \alpha_{min}^1 \leq \alpha_k^1$ et $\alpha_{min}^2 \leq \alpha_k^2$ ou $\alpha_{min}^1 = 1$ et $\alpha_{min}^2 = 1$. Notons $to_{min} = \text{Min}(to_i | i \in I)$. Après α_{min}^1 u.t., le système passe de l'état e^1 à f^1 et après α_{min}^2 u.t. de e^2 à f^2 . Les états f^1 et f^2 ne sont que des états intermédiaires puisqu'aucune action n'a eu lieu. Selon la valeur to_{min} , deux cas se présentent :

1. $to_{min} > 0$: dans ce cas, l'état de r_i pour $i \in I$ devient $\langle s_i, \ell_i, to_i - 1, 1 \rangle$ dans f^1 et f^2 . Les états des autres robots restent inchangés, à l'exception des valeurs résiduelles qui diminuent de α_{min}^1 u.t. dans f^1 et de α_{min}^2 u.t. dans f^2 . Par conséquent, les nouvelles positions relatives des valeurs résiduelles sont identiques dans les deux états f^1 et f^2 . Donc, les deux états intermédiaires f^1 et f^2 sont équivalents. Ainsi de suite, nous répétons les opérations de décrémentation jusqu'à ce que nous arrivions au cas suivant ($to_{min} = 0$). Notons que nous sommes sûrs d'y arriver puisqu'à chaque itération, au moins un to_i est décrémentée et aucun autre n'est incrémenté.
2. $to_{min} = 0$: soit $I' = \{i \in I | to_i = 0\}$. L'ensemble des robots r_i tel que $i \in I'$ est exactement l'ensemble des robots dont les timeouts expirent en même temps dans f^1 et f^2 . Donc, leurs états sont identiques dans f^1 et f^2 . Ils exécuteront soit l'action WAIT, soit l'action MISS. Le choix probabiliste de l'action MISS produit un ensemble de nouveaux états atteints à partir de f^1 (resp. f^2). A titre d'exemple, si k robots exécutent leurs actions MISS, ceci produit 2^k nouveaux états à partir de f^1 (resp. f^2), chacun avec une probabilité de $1/2^k$. Notons g^1 et g^2 deux nouveaux états intermédiaires dans lesquels le résultat du choix uniforme est le même. Alors, certains robots r_i avec $i \in I'$ peuvent exécuter l'action WAIT, puis l'action SYNC avec leurs homologues. Ces derniers ont les mêmes états aussi bien dans g^1 que dans g^2 à l'exception des valeurs de leurs timeouts. Or, la condition de SYNC est indépendante de la valeur du timeout. Par conséquent, l'exécution des actions SYNC à partir des deux états g^1 et g^2 produira deux nouveaux états h^1 et h^2 qui sont choisis parmi 2^k états possibles successeurs de respectivement e^1 et e^2 dans la chaîne de Markov. Il reste à prouver que h^1 et h^2 sont équivalents. Puisque les mêmes actions produisent les mêmes effets, h^1 et h^2 ne peuvent être différents que par les valeurs des timeouts. Examinons tous les cas possibles. Un robot r_i qui a exécuté une seule action, en l'occurrence WAIT, a réarmé son timeout de N dans les deux états h^1 et h^2 ($\alpha_i = 1$). Un robot qui a exécuté au moins une action, SYNC ou MISS, a réarmé son timeout de 1 u.t. dans les deux états h^1 et h^2 ($\alpha_i = 1$). Le timeout d'un robot qui n'a exécuté aucune action est décrémentée de α_{min}^1 dans h^1 et de α_{min}^2 dans h^2 . Par conséquent, les positions relatives des nouvelles valeurs résiduelles sont les mêmes dans h^1 et dans h^2 . ■

Une chaîne de Markov peut être visualisée par un graphe dont les sommets sont les états et les arcs sont les transitions qui s'effectuent entre ces états. Une transition entre un état s et un autre s' est possible si et seulement si la probabilité de transition du premier au second est strictement positive ($P[s, s'] \neq 0$). Le lemme suivant, uniquement valide dans le cas de chaînes finies, facilitera la preuve de correction de notre algorithme.

Lemme 3.5 (Feller, 1968) *Soit S' un sous-ensemble de l'ensemble fini des états d'une chaîne de Markov. Supposons que pour tout état s , il existe un chemin de s à s' tel que $s' \in S'$. Alors quel que soit l'état initial, la probabilité d'atteindre un état appartenant à S' est 1.*

3.5.2 États stables

Dans la suite, nous posons une condition qui nous garantira que, pour toute exécution partant d'un état satisfaisant cette condition, l'occurrence d'une action *MISS* n'est pas possible. Pour cela, nous avons besoin de quelques définitions basées sur la modélisation en réseau de Petri de l'algorithme originel.

Définition 3.3 *Soit $e = \prod_{i=1}^m \langle s_i, \ell_i, to_i, \alpha_i \rangle$ un état du système. Alors le marquage $M(e)$ de N est défini comme suit : si $\ell_i = f(i, j)$ alors $M(e)(p_{i,j}) = 1$, sinon $M(e)(p_{i,j}) = 0$.*

En effet, le marquage $M(e)$ n'est qu'une abstraction de l'état e où les informations de temps et du statut du robot sont omises.

Définition 3.4 *Soit N le réseau de Petri modélisant l'algorithme et M son marquage. M est dit sans blocage si tous les circuits de N sont marqués.*

Dans un état sans blocage, l'exécution de l'algorithme originel garantit qu'aucun blocage n'est possible dans le futur. Cependant, avec les différentes valeurs des timeouts, il se peut que dans un état e avec $M(e)$ sans blocage, une action *MISS* soit exécutée (par exemple, suite à l'expiration du timeout d'un robot attendant son homologue qui n'est toujours pas arrivé). Par conséquent, des contraintes de temps doivent être ajoutées à l'état e .

Si M est sans blocage, alors la relation $aide_M$ introduite dans le lemme 3.1 définit un graphe sans cycle. La longueur d'un chemin étant le nombre d'arcs le constituant, nous définissons $level_M(t)$ comme la longueur du plus long chemin d'extrémité t . Nous présentons, sur la figure 3.5, le niveau des transitions pour le marquage initial du réseau de la figure 3.3. Définissons dès maintenant la condition de stabilité sur les états du système.

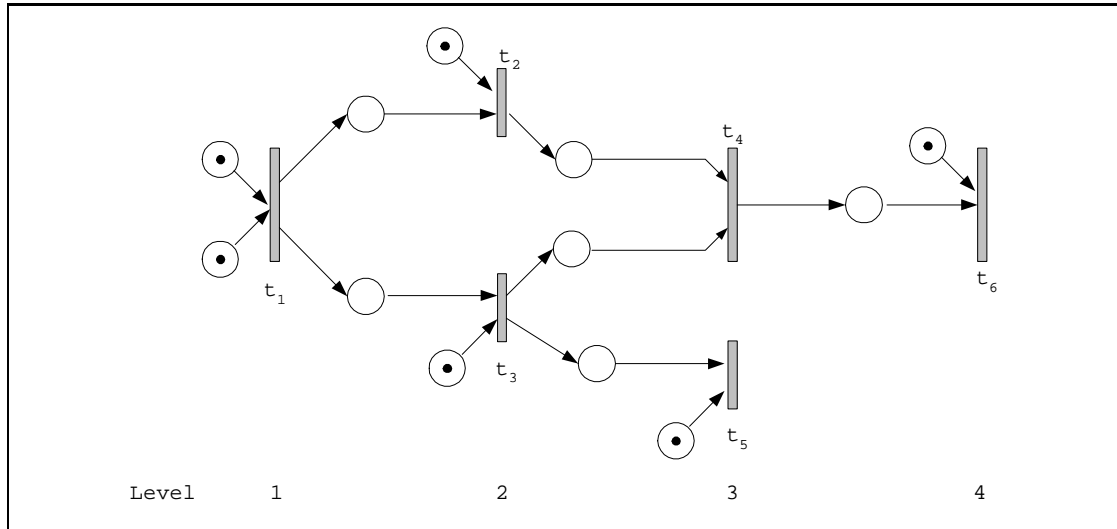


FIG. 3.5 – Niveau des transitions dans un marquage sans blocage

Définition 3.5 Soit $e = \prod_{i=1}^m \langle s_i, \ell_i, to_i, \alpha_i \rangle$ un état du système. e est stable si $M(e)$ est sans blocage et $\forall i, s_i = \text{waiting} \Rightarrow to_i \geq \text{level}_{M(e)}(t_{\ell_i})$ ou $(to_i + 1 > \text{level}_{M(e)}(t_{\ell_i})$ et $\alpha_i = 1$).

Le lemme suivant montre que la définition des états stables est appropriée :

Lemme 3.6 Dans toute exécution partant d'un état stable, aucune occurrence de l'action MISS n'est possible.

Preuve. Nous procédons par récurrence sur les états du système aux instants $0, 1, 2, \dots$. Notons e^n l'état du système à l'instant n et e^0 l'état initial stable du système. Ces états ne correspondent pas aux états successifs de la chaîne de Markov mais cela importe peu puisqu'aucun argument probabiliste ne sera utilisé dans cette preuve. L'hypothèse à vérifier est que jusqu'à l'instant n , aucune action MISS n'aura lieu et que e^n est un état stable.

Pour $n = 0$, aucune action MISS n'a pu avoir lieu et e^0 est stable. Supposons que l'hypothèse est vraie à l'instant n et montrons qu'elle le reste à l'instant $n + 1$. Pour cela, examinons ce qui se passe entre ces deux instants.

Un robot en attente sur un point de rendez-vous à l'instant n , restera sur place à l'instant $n + 1$. En effet, d'après la définition de stabilité, la valeur du timeout de ce robot est supérieure à 1 et par conséquent n'expirera pas en passant à l'instant $n + 1$. Voyons maintenant le cas d'un robot qui vient de se diriger vers un point de rendez-vous à l'instant n . Dans ce cas, il arrive à ce point durant l'intervalle de temps $[n \dots n + 1]$ et réarme son timeout de *Nu.t.*. Donc, à l'instant $n + 1$ soit il est toujours sur place en

attente, soit il exécute l'action SYNC. Dans les deux cas de figure, aucune action MISS n'est exécutée durant cet intervalle de temps.

Il reste à démontrer que e^{n+1} est un état stable. Puisqu'aucune action de type MISS n'est exécutée durant cet intervalle, l'exécution correspond à une exécution de l'algorithme originel sans tenir compte des valeurs des timeouts. Donc, le franchissement d'une séquence de transitions à partir de $M(e^n)$ donne un nouveau marquage $M(e^{n+1})$ dans lequel tous les circuits sont marqués. Soit t une transition de niveau 1 dans $M(e^n)$. Les deux places entrées de t sont marquées soit parce que les deux robots correspondants y sont déjà, soit parce qu'ils sont en déplacement vers ce point commun de rendez-vous. Par conséquent, la synchronisation aura lieu à l'instant $n + 1$.

Discutons du cas d'un robot r_i en attente sur un point de rendez-vous à l'instant $n + 1$. Deux cas de figure se présentent : le robot arrive à la fin ou au début de l'intervalle de temps $[n \dots n + 1]$. Dans le premier cas, il a réarmé son timeout de $Nu.t.$ et par conséquent, à l'instant $n + 1$, la valeur entière de son timeout to_i est égale à $N - 1$ et la valeur résiduelle α_i est égale à 1. Dans ce cas, l'inégalité $to_i + 1 > level_{M(e)}(t_{\ell_i})$ (la borne supérieure des niveaux) et $\alpha_i = 1$ est vérifiée. Dans le second cas, le timeout de r_i est décrémenté d'une unité de temps à l'instant $n + 1$ mais la transition correspondante n'a pas été franchie puisqu'elle était à un niveau supérieur à 1 à l'instant n . Par contre, toutes les transitions de niveau 1 sont franchies à l'instant $n + 1$, et par conséquent le niveau de toutes les autres transitions non franchies est décrémenté de 1. Dans ce cas, l'inégalité $to_i \geq level_{M(e)}(t_{\ell_i})$ reste vérifiée, et par conséquent e^{n+1} est un état stable. ■

3.5.3 D'un état initial à un état stable

Dans cette section, nous montrons qu'à partir d'un état initial quelconque, il existe un chemin allant de cet état vers un état stable dans la chaîne de Markov. Ainsi, la proposition 3.3 découle immédiatement des deux lemmes 3.5 et 3.6. Le seul point non trivial est que pour deux états équivalents quelconques s et s' , s est stable si et seulement si s' est stable puisque la définition de la stabilité ne prend en compte que les valeurs résiduelles entières égales à 1. Par agrégation, tout chemin dont l'état final est stable correspond, dans la chaîne finie agrégée, à un chemin dont l'état final est l'ensemble des états stables.

Lemme 3.7 *A partir d'un état initial quelconque, il existe un chemin dans la chaîne de Markov allant de cet état vers un état stable.*

Preuve. Puisque nous cherchons un chemin particulier dans la chaîne de Markov, nous supposons qu'à chaque exécution de l'action MISS nous fixons l'issue du choix aléatoire.

Dans la suite, nous dirons qu'un robot reproduit le comportement de l'algorithme originel lorsqu'il choisit de rester sur place.

Le cas trivial est celui où l'état initial est stable. Supposons que l'état initial, noté e , ne soit pas stable. Examinons, selon le marquage de e , les deux cas suivants :

1. $M(e)$ est *sans blocage* : l'état e n'étant pas stable, la contrainte de temps n'est pas satisfaite. Nous reproduisons alors le comportement de l'algorithme originel et nous affirmons qu'un état stable sera atteint.

Tout d'abord, tous les marquages successifs seront sans blocage puisqu'ils sont atteints à partir de $M(e)$ dans le réseau N .

Ensuite, nous décomposons le temps en des intervalles égaux de une unité de temps chacun. Durant chaque intervalle, tous les points de rendez-vous correspondant aux transitions de niveau 1 feront l'objet des actions SYNC. Un robot qui exécutera une telle action satisfera la contrainte de temps puisqu'il sera en déplacement. Un raisonnement analogue à celui de la preuve du lemme 3.6 permet de montrer qu'une fois la contrainte de temps satisfaite pour un robot, elle le reste par la suite. Par conséquent, l'état atteint après que chacun des robots ait exécuté au moins une action SYNC est stable.

2. $M(e)$ est *avec blocage* : puisqu'il existe une séquence de points de rendez-vous reliant n'importe quelle paire de robots, la reproduction de l'algorithme originel aboutira à un blocage total. Ainsi, nous reproduisons l'algorithme originel jusqu'au point où chaque robot se trouve seul sur un point de rendez-vous et a exécuté au moins une fois l'action MISS. Dans ce cas, tous les timeouts ont des valeurs inférieures ou égales à 1. À partir de ce moment, chaque robot choisira la seconde alternative durant l'exécution de son action MISS. Toutes ses actions s'exécuteront en moins de *1u.t.*. Après la dernière exécution d'une action MISS, chacun des robots est sur le chemin de son premier point de rendez-vous. Dans un tel état, noté e' , le marquage $M(e')$ correspond au marquage initial du réseau N de l'algorithme originel. Par conséquent, $M(e')$ est sans blocage et la suite du chemin est complétée par le chemin généré par le cas précédent. ■

3.6 Conclusion

Dans ce chapitre, nous avons étendu au systèmes de robots mobiles le domaine d'application de l'auto-stabilisation, jusque-là réservée aux systèmes filaire. Nous avons présenté un algorithme d'ordonnancement non auto-stabilisant. Afin d'établir un ordonnancement,

une numérotation arbitraire des points de rendez-vous est choisie et chaque robot parcourt ses points de rendez-vous dans l'ordre croissant de leurs numéros. Il s'agit de partir du point de rendez-vous ayant le plus petit numéro, puis de visiter tous les autres points de manière unidirectionnelle et répétitive. Nous avons modélisé le comportement de chaque robot par un réseau de Pétri local et le comportement du système par un réseau de Pétri global obtenu par fusion des transitions. Le graphe global obtenu étant un graphe d'événements, nous avons utilisé des propriétés propres à cette famille de graphes pour prouver l'absence de blocage partiel et global de l'algorithme.

Ensuite, nous avons décrit formellement la transformation de l'algorithme précédent en un algorithme auto-stabilisant sous un démon probabiliste. Cet algorithme, basé sur une stratégie de visites des points de rendez-vous, assure qu'après la phase de stabilisation chaque visite à un point de rendez-vous aboutit à une communication. Le comportement d'un robot est événementiel : l'occurrence d'un événement déclenche l'exécution d'un code dépendant de l'état courant du robot. Le comportement de ce nouvel algorithme se modélise par une chaîne de Markov à temps discret. Les résultats théoriques de ce domaine, tels que les conditions d'agrégation forte, nous ont servi à modéliser le comportement de l'algorithme en une chaîne de Markov *finie* et à établir la preuve de stabilisation de l'algorithme. Pour cela, nous avons défini une condition suffisante de stabilité sur les états du système et nous avons montré qu'à partir d'une classe d'états initiaux quelconques, il existe un chemin allant de cette classe vers la classe d'états stables dans la chaîne de Markov agrégée.

Dans le chapitre suivant, nous abordons un autre type de protocole de communication nécessaire dans les systèmes de robots mobiles : le routage.

Chapitre 4

Routage auto-stabilisant¹

Résumé. *Ce chapitre est consacré au problème de routage dans un système de robots mobiles. Nous nous intéressons tout d'abord au graphe de communication des points de rendez-vous. Pour le construire, on duplique chaque point de rendez-vous, partagé par deux robots, en deux nœuds du graphe, correspondant à chacun de ces robots. Un lien reliant ces deux nœuds correspond à la communication entre les deux robots concernés. Dans un tel système, une table de routage d'un robot doit comporter, pour être exploitable, une entrée pour tout couple (point de rendez-vous d'un robot, destination). En effet, le plus court chemin dépend du point de rendez-vous sur lequel le robot émetteur est positionné. Nous proposons ensuite un algorithme auto-stabilisant, qui établit un routage entre les robots distants. Cet algorithme suppose que l'algorithme d'ordonnancement décrit dans le chapitre 3 soit stabilisé, et que toute visite à un point de rendez-vous aboutisse à une communication. Nous établissons alors la preuve de son auto-stabilisation en examinant l'évolution de la forêt d'arborescences des plus courts chemins construite par l'algorithme.*

¹Ce chapitre s'appuie en partie sur des résultats présentés dans El Haddad et Haddad (2004b).

4.1 Introduction

Sur un réseau, il peut exister plusieurs chemins possibles permettant de faire transiter l'information jusqu'à sa destination. Le routage est la tâche qui consiste à transférer les messages en fonction des informations contenues dans la table de routage. Les protocoles de routage sont des programmes qui échangent des informations utilisées afin de créer les tables de routage. Tous les protocoles de routage exécutent les mêmes fonctions de base. Ils déterminent la "meilleure" route vers chaque destination et distribuent les informations d'acheminement entre les processeurs du système. Les modalités d'exécution de ces fonctions, en particulier les procédures de sélection des meilleures routes, appelé *métriques*, permettent de distinguer les différents protocoles. Chaque métrique représente un élément d'information permettant au système de choisir la meilleure route. Il peut s'agir d'un nombre de tronçons (de l'anglais *hop count*), d'un délai d'acheminement ou encore d'une valeur arbitraire définie par l'administrateur. Pour toutes ces métriques, la spécification d'une valeur basse est préférable à celle d'une valeur élevée. A titre d'exemple, le protocole de routage RIP sélectionne la route offrant *le nombre de tronçons le plus petit*. Nous rappelons brièvement ci-dessous certains algorithmes de routage qui ont été proposés dans le domaine de l'auto-stabilisation.

Bref état de l'art. En 1997, Dolev a présenté un protocole de routage auto-stabilisant basé sur la métrique *minimiser le nombre de routeurs dans les chemins*. Cette métrique est similaire à la métrique *plus court chemin* dans le cas où tous les liens de communications ont le même coût. Pour cette métrique, Arora et al. (1990) ont présentés deux protocoles auto-stabilisants de routage. De même, Gouda et Schneider (1999) ont présenté un protocole auto-stabilisant générique de construction de tables de routage dans des réseaux uniformes. Pour la métrique du débit maximal, Gouda et Schneider (2003) ont présenté un protocole auto-stabilisant pour construire et maintenir des tables de routage dans des graphes de topologie quelconque. Plus récemment, Cobb et Gouda (2002) ont présenté un protocole auto-stabilisant générique de construction de tables de routage tolérant le changement des "valeurs" des canaux. Pour ce faire, les processeurs doivent connaître une borne supérieure sur la taille du réseau.

Pour des systèmes à grande échelle, le routage hiérarchique permet de réduire la taille des tables de routage et donc le temps nécessaire pour router les messages. Un protocole auto-stabilisant de routage hiérarchique pour la métrique des plus courts chemins a été présenté par Datta et al. (2000). Pour le routage inter-domaine, un algorithme auto-stabilisant a été proposé par Chen et al. (2002).

Pour des systèmes sans fil, Nesterenko et Arora (2001) ont proposé deux protocoles auto-stabilisants de routage pour un système asynchrone et semi-uniforme en utilisant

deux techniques opposées. La première consiste à détruire et à reconstruire périodiquement l'arbre des plus court chemins. La seconde consiste à construire progressivement l'arbre pour la métrique *le plus petit nombre de tronçons*. Plus récemment, dans le cadre d'un modèle à registres, Arora et Zhang (2003) ont proposé un algorithme auto-stabilisant et confinant les fautes pour des systèmes uniformes. Cet algorithme est composé de trois vagues : la vague de stabilisation, la vague de confinement et la vague de super-confinement de fautes. Chacune de ces vagues avance à une vitesse différente, la plus lente étant la vague de stabilisation. Cette dernière n'est autre qu'une adaptation de l'algorithme de plus court chemin de Bellman-Ford (e.g., Tel, 1994), lui permettant de coopérer avec l'algorithme de confinement de fautes. Elle permet de propager les valeurs correctes afin que le système converge vers un état légitime. Cependant, une telle vague, initiée par un processeur fautif, propage de fausses valeurs dans le système. Pour remédier à cela, la vague de confinement de fautes a été introduite. Elle se propage sur les chemins établis par la vague de stabilisation, mais plus rapidement de façon à rattraper une vague initiée par un processeur fautif et à arrêter ainsi la propagation de fausses valeurs. De même, pour remédier à une vague de confinement de fautes initiée par erreur, la vague de super-confinement de fautes a été introduite. Elle se propage sur les mêmes chemins que la vague de confinement de faute mais plus rapidement de façon à la rattraper et à l'arrêter.

La communication entre robots étant supposée sûre, la mobilité pose le problème de l'acheminement des messages. En effet, la position d'un robot au moment d'émettre un message est un paramètre important. Notre contribution consiste à proposer un algorithme auto-stabilisant de routage pour un système asynchrone et semi-uniforme de robots mobiles. Cet algorithme repose sur la métrique *le plus court chemin* dans le cadre d'un modèle à passage de messages.

4.2 Routage auto-stabilisant dans les systèmes de robots mobiles

Après avoir produit un ordonnancement des déplacements des robots dans le chapitre 3, nous nous intéressons au routage dans un tel système. Nous décrivons un algorithme auto-stabilisant, adapté de celui de Dolev, Israeli et Moran (1989), responsable de la génération et de la maintenance des tables de routage des robots.

4.2.1 Le modèle

Pour calculer les distances correspondant aux plus courts chemins, nous construisons le graphe de communication du système à partir des points de rendez-vous. Un point

de rendez-vous étant partagé par deux robots, nous lui associons deux noeuds dans le graphe, un pour chaque robot. Ainsi, un noeud de ce graphe est un couple (r_i, mp) dont le premier élément est l'identité d'un robot et le second élément est l'identifiant d'un de ses points de rendez-vous. Dans ce graphe orienté, deux types d'arcs existent :

- le premier type d'arc relie un noeud (r_i, mp) à un noeud (r_i, mp') si il existe un j tel que $mp = MP_i[j]$ et $mp' = MP[(j + 1) \bmod n_i]$;
- Le second type d'arc relie un noeud (r_i, mp) à un noeud (r_j, mp) . Ce dédoublement des points de rendez-vous du système augmente le nombre des liens du graphe de communication.

Remarquons que tout arc du premier type, représentant le déplacement d'un point de rendez-vous à un autre, augmente la distance de 1. De même, tout arc du second type, représentant l'échange de message entre un robot r_i et son homologue r_j , augmente la distance de 1. Par conséquent, la valeur d'une distance dans un tel graphe ne désigne pas la distance réelle mais une distance que nous appelons virtuelle puisque nous supposons que l'échange de messages sur un point de rendez-vous est instantané. A titre d'exemple, le graphe de communication du système décrit dans l'exemple 3.1 est présenté sur la figure 4.1.

Une entrée dans la table de routage d'un robot est indiquée par un couple (point de rendez-vous, identité du robot destination). Cette entrée indique au robot, lorsqu'il arrive au point de rendez-vous correspondant, s'il doit transmettre ou non à son homologue les messages de la destination correspondante. En effet, le chemin le plus court vers une destination dépend de la position du robot au moment où il décide d'émettre un message. L'algorithme que nous proposons construit, à partir d'un point de rendez-vous fixé d'un robot, les plus courts chemins à l'ensemble des points de rendez-vous des autres robots.

Pour simplifier la présentation, nous nous restreignons à une destination particulière, la généralisation étant immédiate. Nous considérons r_0 , le robot à atteindre. En plus des constantes et des variables précédentes, chaque robot possède une identité unique r_i placée dans sa mémoire incorruptible, ainsi que les variables suivantes nécessaires à la construction et au maintien de sa table de routage :

- $distance_i[0 \dots n_i - 1]$: un élément $distance_i[j]$ de ce tableau est l'estimation courante par r_i de la distance minimale séparant son point de rendez-vous $MP_i[j]$ de r_0 . L'intervalle des valeurs possible pour cette variable est $[0 \dots 2N - 1]$, où N est le nombre total des points de rendez-vous dans le système.
- $sortie_i[0 \dots n_i - 1]$: tableau de booléens indicé par les points de rendez-vous de r_i . Quand $sortie_i[j]$ est à *vrai* et que r_i arrive au point de rendez-vous, il passera les

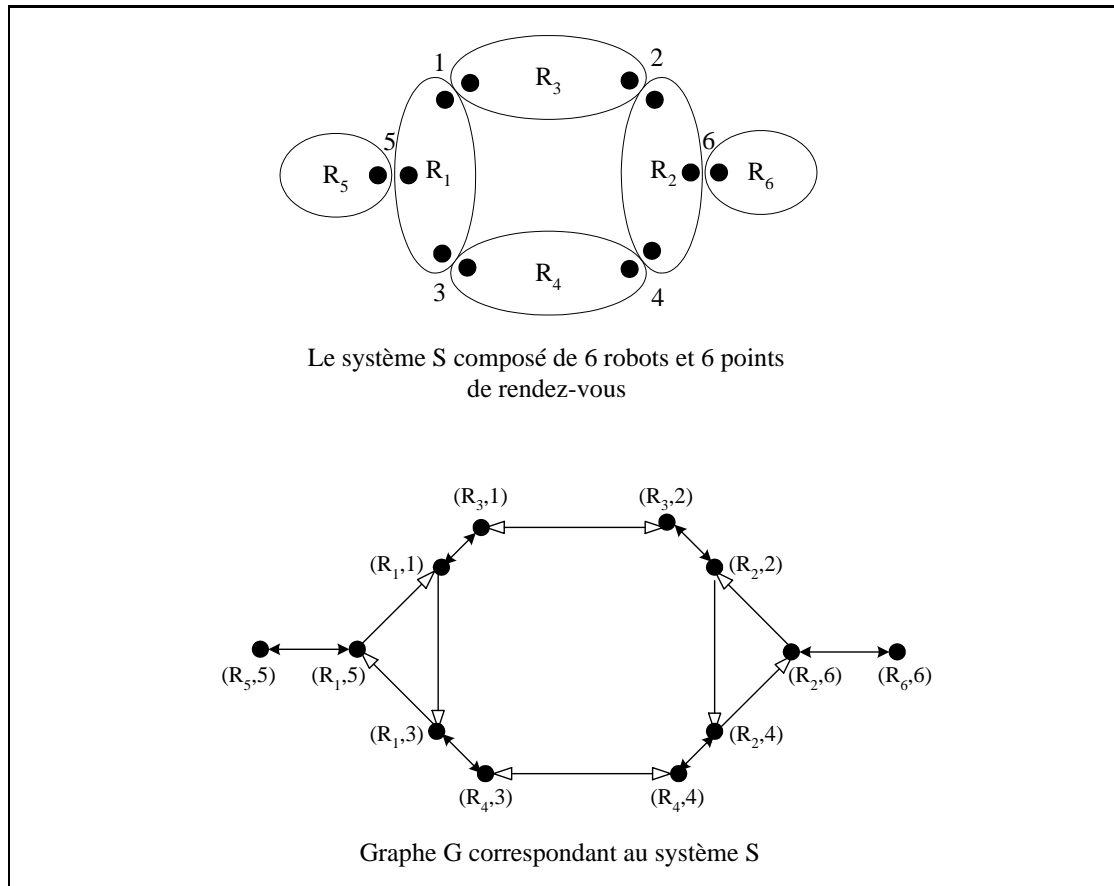


FIG. 4.1 – Graphe des points de rendez-vous

messages à destination de r_0 à son homologue via ce point de rendez-vous commun. Notons qu'un robot peut avoir plusieurs attributs *sortie* positionnées à vrai.

Plus précisément, une entrée dans la table de routage d'un robot est constituée de 4 champs :

1. la position du robot émetteur du message,
2. l'identité du robot destination,
3. la distance correspondant au plus court chemin entre le robot émetteur (se trouvant sur la position contenue dans le premier champ) et le robot destination,
4. le point de rendez-vous suivant sur le plus court chemin vers la destination déduit de la valeur de l'attribut *sortie*.

Exemple 4.1 *Considérons le graphe de la figure 4.1, les entrées de la table de routage de r_1 vers la destination r_6 sont les suivantes :*

| | <i>distance</i> ₁ | <i>sortie</i> ₁ |
|------------|------------------------------|----------------------------|
| $(r_1, 1)$ | 6 | <i>vrai</i> |
| $(r_1, 3)$ | 5 | <i>vrai</i> |
| $(r_1, 5)$ | 7 | <i>faux</i> |

TAB. 4.1 – Les entrées de la table de routage de r_1 à destination de r_6

Dans cette table, si le robot r_1 désire envoyer un message au robot r_6 , il choisira de l'envoyer soit via r_3 s'il est aux points de rendez-vous 1 ou 5, soit via r_4 s'il se trouve sur le point de rendez-vous 3. Remarquons que l'algorithme trouve le plus court chemin entre les points de rendez-vous d'un robot r_i et l'un quelconque du robot destination.

4.2.2 Algorithme

Dans la suite, nous utilisons la technique de composition équitable introduite par Dovlev et Herman (1999). L'idée consiste à composer deux algorithmes auto-stabilisants, l'un d'eux utilisant le comportement stabilisé de l'autre comme postulat de départ. Nous appliquons cette technique aux algorithmes d'ordonnancement et de routage. Chaque robot exécute en parallèle une action de chacun de ces deux algorithmes. L'algorithme de routage suppose que celui d'ordonnancement s'est stabilisé, et que chaque visite à un point de rendez-vous aboutit à une communication point-à-point durant laquelle les deux robots concernés échangent leur estimation de la distance les séparant du robot destination. Dans le cas contraire, autrement dit si l'algorithme d'ordonnancement n'est pas stable, alors l'algorithme de routage ne garantit pas la correction des entrées des tables de routage.

L'algorithme de routage que nous proposons est donné ci-après (Algorithme 2). Si ce code est exécuté par le robot destination r_0 , alors ce dernier fixe les distances de tous ses points de rendez-vous à 0. Par conséquent, les valeurs du champ *distance* dans la table de routage de r_0 sont correctes après une seule itération. Pour un robot $r_i \neq r_0$, une entrée associée à un point de rendez-vous $MP_i[j]$ dans sa table de routage est mise à jour comme suit lors de la synchronisation avec son homologue r_k :

- r_i et r_k échangent leurs estimations courantes de la distance les séparant de r_0 en partant de $MP_i[j]$,
- à la réception de la distance courante d de son homologue, r_i calcule sa nouvelle estimation en prenant la valeur minimale entre $d + 1$ et la valeur incrémentée de 1 de la distance associée au point de rendez-vous suivant. Si la distance minimale est celle proposé par son homologue, alors r_i fixe la sortie correspondante *sortie* _{i} [j] à *vrai*.

Algorithme 2 *Algorithme de routage auto-stabilisant - Programme d'un robot r_i*

Variables : $distance_i[1 \dots n_i] \in [0 \dots 2N - 1]$;
 $sortie_i[1 \dots n_i]$;

Pour le robot r_0

répéter périodiquement

pour $j = 0$ à $n_0 - 1$ **faire** $distance_0[j] = 0$;

finrépéter

Synchronisation avec le robot r' sur le point de rendez-vous $MP_i[j]$

 Envoyer_à (r' , $distance_i[j]$) ;

 Recevoir_de (r' , d) ;

si ($r_i \neq r_0$) **alors**

$d' = \min(d, distance_i[(j + 1) \bmod n_i])$;

si ($d' < 2N - 1$) **alors**

$distance_i[j] = d' + 1$;

$sortie_i[j] = (d == d')$;

fin

fin

fin

Exemple 4.2 *Considérons le système présenté sur la figure 4.1, composé de 6 robots et 6 points de rendez-vous. L'objectif de cet exemple est de montrer une exécution possible de l'algorithme 2 avec le robot r_6 comme destination. La configuration initiale est telle que le robot r_6 ait mis à 0 la valeur de la distance de son point de rendez-vous d'identifiant 6 et à faux sa variable sortie. La configuration initiale est présentée sur la figure 4.2. Chaque robot r_i dispose d'une variable distance à valeur dans $[0 \dots 2N - 1]$ pour chaque point de rendez-vous. Cette variable représente la distance du plus court chemin séparant ce point de rendez-vous du robot destination r_6 . Une exécution possible de l'algorithme de routage est décrite sur la figure 4.3. L'exécution est divisée en rondes. Durant une ronde, chaque robot fait un tour complet de ses points de rendez-vous. Sur chacun de ses points, un robot r_i calcule la nouvelle estimation de la distance ainsi que la nouvelle valeur de sa variable sortie (représentée par un arc fléché). Remarquons que les points de rendez-vous se stabilise de proche en proche en partant des points de rendez-vous du robot destination. En effet, après la première ronde, l'ensemble des noeuds à distance inférieure ou égale à 1 est organisé en une arborescence des plus courts chemins. Ainsi de suite, après la huitième ronde, l'ensemble des noeuds à distance inférieure ou égale à 8 est organisé en une arborescence des plus courts chemins.*

4.3 Preuve de stabilisation

Rappelons que pour calculer les distances correspondant aux plus courts chemins, nous avons construit un graphe G de communication dont les noeuds sont des couples

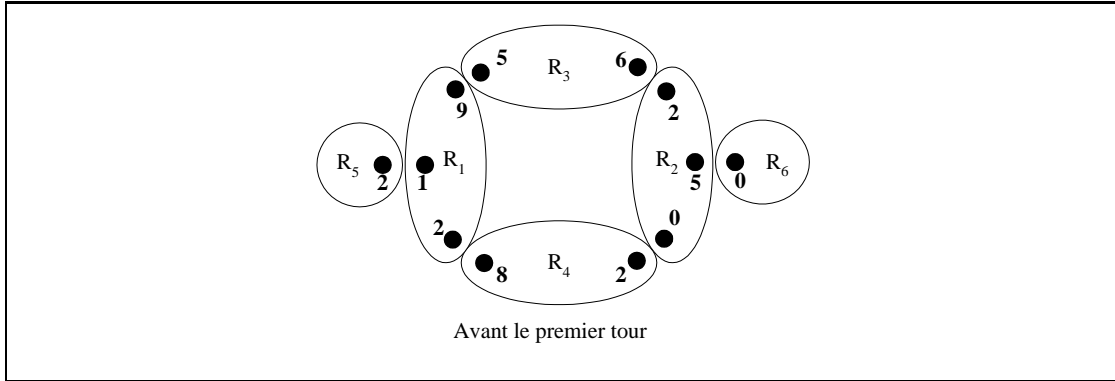


FIG. 4.2 – Configuration initiale

(r_i, mp) . Dans la suite, nous définissons un sous-graphe SG comme un graphe dont les noeuds sont les mêmes que ceux de G et possédant deux types d’arcs :

- les arcs reliant un noeud (r_i, mp) à un noeud (r_i, mp') si et seulement si il existe un j tel que $mp = MP_i[j]$, $mp' = MP[(j + 1) \bmod n_i]$ et $sortie_i[j] = faux$,
- les arcs reliant un noeud (r_i, mp) à un noeud (r_j, mp) si et seulement si il existe un j tel que $mp = MP_i[j]$ et $sortie_i[j] = vrai$.

Dans cette section, nous prouvons que le sous-graphe orienté se stabilise en formant une forêt d’arborescences dont les racines sont les points de rendez-vous du robot destination. A titre d’exemple, le sous-graphe correspondant au graphe de la figure 4.1 est présenté sur la figure 4.4. Dans la suite, nous formalisons cet argument pour prouver la correction de l’algorithme. La structure d’une arborescence est déduite des variables $distance_i[j]$ et $sortie_i[j]$ pour tous les noeuds $(r_i, MP_i[j])$ appartenant à l’arborescence.

Nous introduisons quelques concepts associés à une exécution. Une exécution se divise en rondes successives. Chaque ronde débute à la fin de la ronde précédente et se termine lorsque tous les robots ont réalisé au moins un tour complet de leurs points de rendez-vous. La première ronde débute après que l’algorithme d’ordonnancement précédent soit stabilisé et que le robot destination r_0 ait fixé les distances de tous ses points de rendez-vous à 0.

Définition 4.1 *Étant donné une configuration, une “valeur fausse” est une distance assignée à une variable et ne correspondant pas à la distance réelle.*

Notons ppv_f la plus petite valeur fausse présente dans une configuration. S’il n’existe pas de valeur fausse dans une configuration, alors nous posons $ppv_f = \infty$. Le lemme suivant établit la propriété-clef de notre algorithme de routage :

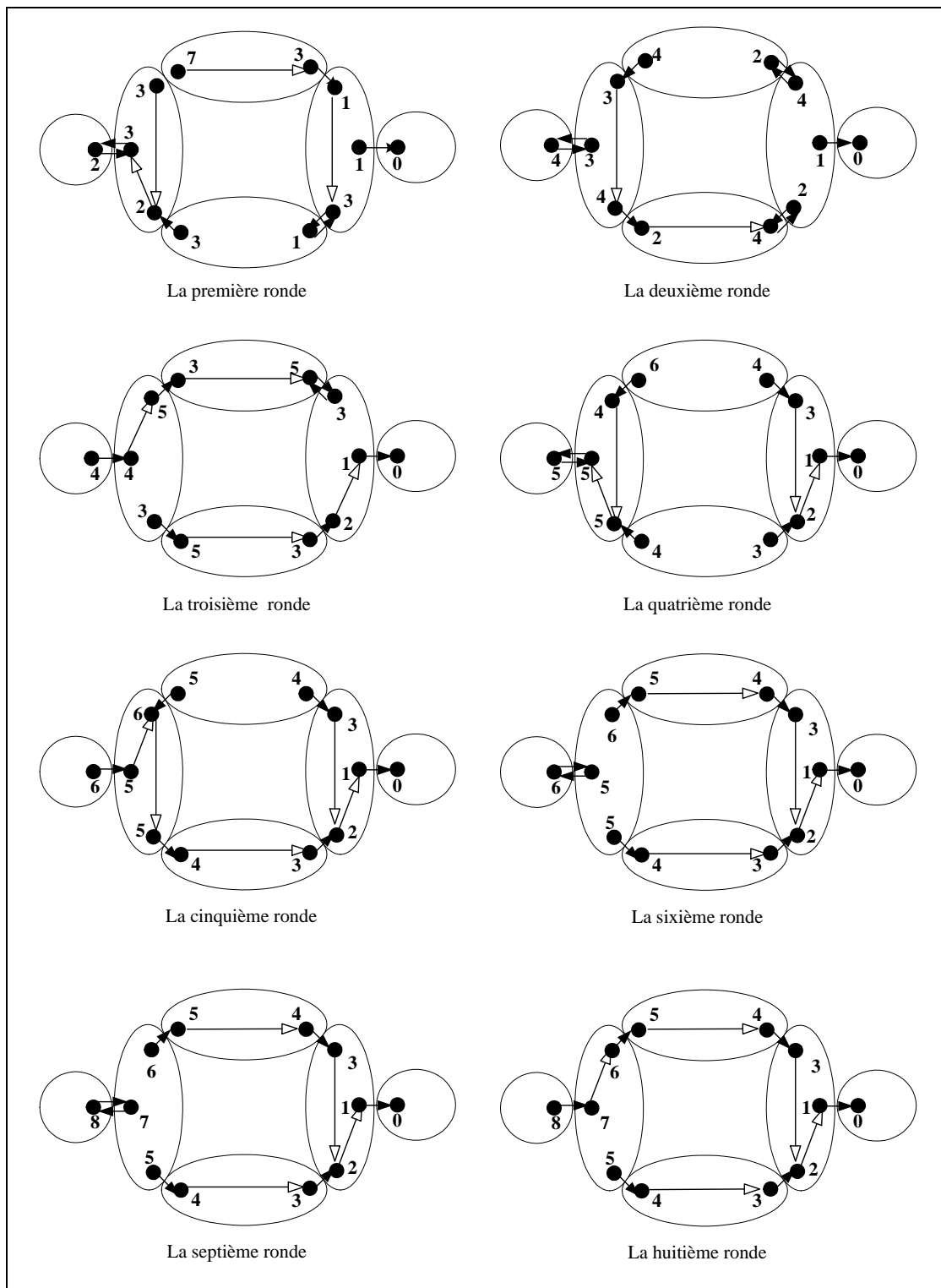
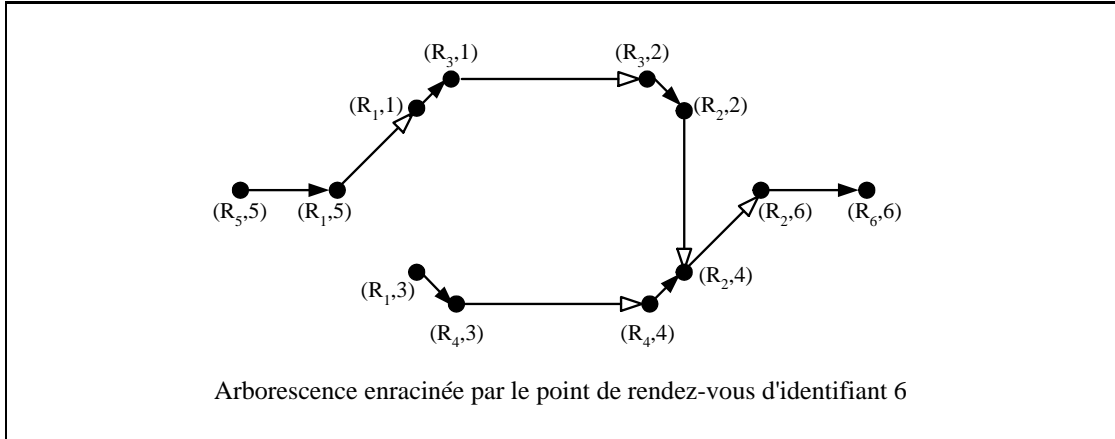


FIG. 4.3 – Une exécution possible de l'algorithme de routage

FIG. 4.4 – Le Sous-graphe SG stabilisé

Lemme 4.1 *Pour tout $k \geq 0$ et pour toute configuration qui suit les premières k rondes, nous avons :*

1. $ppv f \geq k$;
2. *l'ensemble des noeuds à distance inférieure ou égale à k est organisé en une forêt d'arborescences des plus courts chemins.*

Preuve. Notons $ppv f^k$, la $ppv f$ à la fin de la $k^{\text{ième}}$ ronde et $ppv f^0$ la $ppv f$ initiale. Nous allons prouver le lemme par récurrence sur k .

Pour $k = 1$, d'après la définition de la distance, toutes les valeurs contenues dans toutes les variables distances sont strictement positives. Par conséquent, $ppv f^0 \geq 0$. Durant la première ronde, chaque robot fait le tour de ses points de rendez-vous et calcule toutes les distances respectives. Durant cette ronde, un robot voisin r_i recevant une distance $ppv f^0$ d'un de ses voisins r_j sur le point de rendez-vous $MP_i[j]$, re-calculera sa distance. S'il trouve que $ppv f^0$ est une distance minimale, alors la nouvelle valeur de $distance_i[j]$ est $ppv f^0 + 1$. Ainsi, suite à la première ronde, aucune des variables du tableau $distance_i$ d'un robot r_i différent de r_0 ne porte une valeur égale à 0. Ceci prouve qu'après la première ronde $ppv f^1 \geq 1$.

D'autre part, la première ronde débute après que l'algorithme d'ordonnancement précédent soit stabilisé et que le robot destination r_0 ait fixé les distances de tous ses points de rendez-vous à 0. Autrement dit, avant le début de la première ronde, tous les noeuds $(r_0, MP_0[j])$ du sous-graphe SG ont leurs variables $distance_0[j]$ fixées à 0 et leurs variables $sortie_0[j]$ fixées à *faux*. Ces valeurs sont correctes et ne changeront pas au cours des rondes suivantes. Par conséquent, chaque noeud $(r_0, MP_i[j])$ constitue une racine d'une arborescence. En outre, chaque robot r_j partageant un point de rendez-vous $MP_j[0]$ avec le robot r_0 , recevra durant cette première ronde la distance 0 de r_0 . Puisque $ppv f^0 \geq 0$, la nouvelle

valeur du noeud $(r_j, MP_j[0])$ est égale à 1 et $sortie_j[0]$ est fixée à *vrai*. Ainsi, l'ensemble des noeuds à distance inférieure ou égale à 1 est organisé en une forêt d'arborescences.

Supposons que les propriétés sont valides pour la $k^{\text{ième}}$ ronde et montrons qu'elles restent pour la $(k + 1)^{\text{ième}}$ ronde. En effet, au cours de la $(k + 1)^{\text{ième}}$ ronde, la distance recalculée d'un noeud $(r_i, MP_i[j])$ est soit exacte, soit supérieure ou égale à $ppvf^k + 1$. En effet, considérons un noeud $(r_i, MP_i[j])$ voisin d'un noeud $(r_j, MP_j[i])$ ayant $ppvf^k$ comme valeur de sa distance (voir figure 4.5). Au cours de la $(k + 1)^{\text{ième}}$ ronde, le robot r_i recalcule sa distance sur le noeud $(r_i, MP_i[j])$. La nouvelle valeur de $distance_i[j]$ est donc soit correcte soit égale à $ppvf^k + 1$. De même, le robot r_j recalcule la distance du noeud $(r_j, MP_j[i])$ qui est soit égale à $distance_i[j] + 1$, soit égale à $distance_j[(i + 1) \bmod n_j] + 1$ avec $distance_j[(i + 1) \bmod n_j]$ pouvant être égale à $ppvf^k$. Dans les deux cas, $ppvf^k$ est incrémentée à la fin de la $(k + 1)^{\text{ième}}$ ronde. Ceci prouve que $ppvf \geq k + 1$ après la $(k + 1)^{\text{ième}}$ ronde.

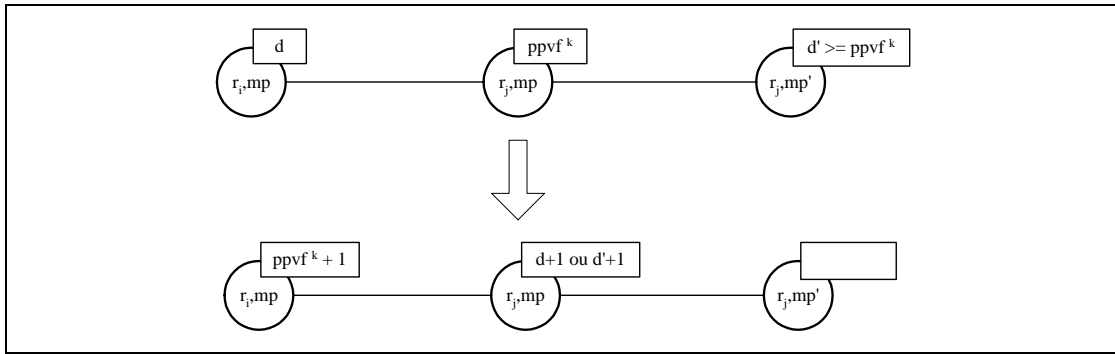


FIG. 4.5 – Après la $(k + 1)^{\text{ième}}$ ronde $ppvf \geq k + 1$

Supposons la deuxième propriété du lemme établie à la $k^{\text{ième}}$ ronde et examinons la $(k + 1)^{\text{ième}}$ ronde. Au cours de cette ronde, la $ppvf$ est supérieure ou égale à k . Donc les noeuds à distance inférieure ou égale à k conservent leurs attributs *distance* et *sortie* inchangés. Un noeud $(r_i, MP_i[j])$ voisin d'un noeud de distance égale à k recalcule sa distance au cours de cette ronde. Il lui affecte la valeur minimale entre $k + 1$ et la valeur incrémentée de 1 de la distance associée au noeud suivant en passant par un arc du premier type (i.e., $distance_i[(j + 1) \bmod n_i] + 1$, voir figure 4.6). Soit la distance associée au noeud suivant est exacte et correspond alors à un noeud à distance k dans le graphe de communication, soit elle est fautive et a alors une valeur $> k$. Dans les deux cas, la distance recalculée associée au noeud $(r_i, MP_i[j])$ est égale à $k + 1$ et ce noeud rejoint l'une des arborescences de la forêt. ■

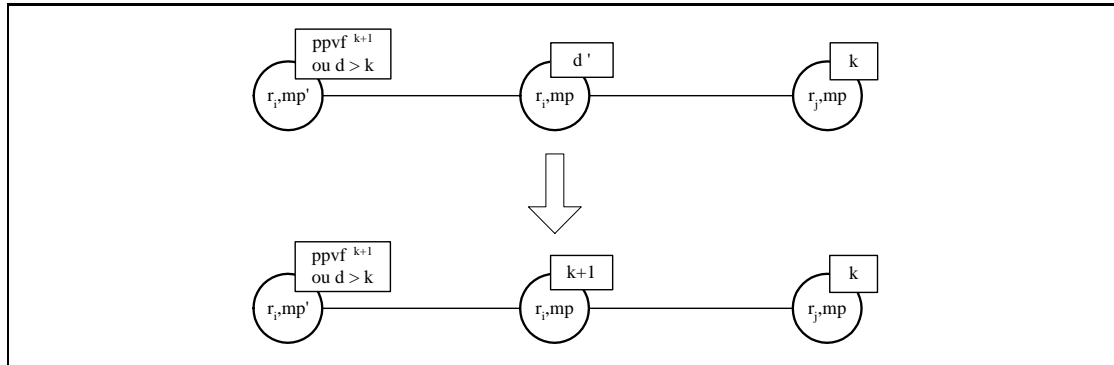


FIG. 4.6 – Les mise à jour des noeuds à distance égale à $k + 1$ durant la $(k + 1)^{i\grave{e}me}$ ronde

Le corollaire suivant découle immédiatement du lemme 4.1.

Corollaire 4.2 *Au bout de $2N$ rondes, le routage est stabilisé dans tout le système.*

4.4 Conclusion

Dans ce chapitre, nous nous sommes intéressés au problème de routage dans des réseaux de robots mobiles avec points de rendez-vous fixes. Nous avons ramené ce problème à la recherche de plus courts chemins dans un graphe. Pour ce faire, nous avons tout d'abord construit le graphe de communication induit par l'ordonnancement des déplacements des robots. Dans ce graphe, un point de rendez-vous correspond à deux nœuds, un pour chaque robot impliqué dans le rendez-vous. L'algorithme que nous avons proposé est une adaptation de celui de Dolev, Israeli et Moran (1989), et suppose que toute visite à un point de rendez-vous aboutit à une communication. Il construit, à partir d'un point de rendez-vous fixé d'un robot, les plus courts chemins allant vers n'importe quel autre point de rendez-vous. Ces plus courts chemins constituent les tables de routage. Une entrée dans la table de routage d'un robot est indiquée par un couple (point de rendez-vous, robot destination). Cette entrée indique, lorsque le robot se trouve sur un point de rendez-vous, s'il doit ou non transmettre le message à son homologue. La preuve de stabilisation de l'algorithme consiste à montrer d'une part la disparition des valeurs erronées éventuellement contenue dans les variables de distance, d'autre part la construction d'une forêt d'arborences couvrant l'ensemble des nœuds du graphe de communication.

Chapitre 5

Confinement de fautes pour le problème de l'élection et de l'arbre couvrant¹

Résumé. *Ce chapitre est consacré au confinement de fautes dans le problème de l'élection et de l'arbre couvrant. Nous nous sommes focalisés sur l'algorithme de Afek et al. (1990) utilisant des prédicats locaux pour une détection des défaillances globales. Il permet de résoudre le problème de l'élection en construisant un arbre qui couvre tous les processeurs du réseau et dont la racine est le processeur d'identité maximale. Après avoir introduit formellement cet algorithme, nous en présentons une variante dotée d'un mécanisme de confinement de fautes. Nous précisons une hypothèse nécessaire pour la conception de l'algorithme : un processeur peut lire les variables des voisins de ses voisins. Sous cette hypothèse, nous construisons l'algorithme de façon à ce que seuls le processeur fautif et ses voisins soient susceptibles d'exécuter des règles. Nous prouvons la propriété de confinement de fautes en montrant que la propagation d'une faute singulière se réduit uniquement aux voisins du processeur fautif. Nous prouvons enfin l'auto-stabilisation de l'algorithme en montrant :*

- la disparition des fausses identités,*
- le maintien des requêtes,*
- la couverture par l'arbre de tous les processeurs du système.*

¹Ce chapitre s'appuie en partie sur des résultats présentés dans El Haddad et Haddad (2004a).

5.1 Introduction

En vue de contrôler un système, il est souvent nécessaire d'amener les processeurs à s'entendre sur une valeur commune ou à se synchroniser. Parmi les problèmes à résoudre pour atteindre cet objectif, on trouve le problème de l'élection d'un leader et les protocoles à vagues. Le problème de l'élection consiste à ce qu'un ensemble de processeurs choisissent un des leurs comme leader. Ce problème est souvent lié au problème de la construction d'un arbre couvrant. Usuellement, le processeur élu est la racine de l'arbre couvrant. Les protocoles à vagues sont employés dans la résolution de nombreux problèmes de synchronisation comme l'allocation de ressources ou la diffusion d'informations. Les deux familles de protocoles à vagues les plus utilisés sont la *circulation d'un jeton* et la *propagation d'information avec retour*. Plusieurs protocoles de propagation d'information avec retour sur des arbres couvrants, à la fois optimaux en espace et instantanément stabilisants, ont été proposés par Bui et al. (1999a,b). Dans ces protocoles, lorsqu'un processeur décide d'envoyer un message à tous les autres processeurs du réseau en initialisant une vague de propagation d'information avec retour, il est sûr que lorsque la vague se termine, tous les processeurs auront bien reçu son message. Dans ce qui suit, nous présentons certains des travaux menés en auto-stabilisation pour résoudre le problème de l'élection.

Bref état de l'art. De nombreux travaux ont montré l'importance des hypothèses qu'on pouvait faire sur le modèle, comme la présence d'identité ou le type de démon, dans la conception des algorithmes. Nous présentons ici les principaux résultats dans le domaine, selon les différentes hypothèses faites sur le modèle. Dans les systèmes avec identifiants, il existe plusieurs protocoles déterministes auto-stabilisants, comme ceux proposés par Afek et al. (1990), Aggarwal et Kuttan (1993) et Arora et Gouda (1994) pour lesquels l'espace mémoire alloué dépend de la taille du réseau. Sur un anneau anonyme, un algorithme d'élection peut être conçu uniquement si l'anneau est de taille première et que le démon est centralisé (e.g., Angluin, 1980; Burns et Pachl, 1989). Toujours pour le même problème, Beauquier, Gradinariu et Johnen (1999b) ont prouvé qu'un algorithme déterministe auto-stabilisant d'élection pour les anneaux unidirectionnels nécessite au moins $\log_2(N)$ bits sur chaque élément du système (N étant la taille du système). Notons que les algorithmes fonctionnant sur des systèmes anonymes sont en général probabilistes : Beauquier, Gradinariu et Johnen (1999c) ont présenté un algorithme auto-stabilisant d'élection sur un anneau et Dolev, Israeli et Moran (1997b) un algorithme auto-stabilisant pour des systèmes à la topologie quelconque. Plus récemment, dans le cadre des systèmes sans fil, Baala et al. (2003) ont proposé un algorithme auto-stabilisant de construction d'arbre couvrant. Pour plus de détails sur les résultats obtenus pour la résolution du problème de l'élection et de l'arbre couvrant, le lecteur pourra se reporter aux articles de

synthèse de Gärtner (2003) et Johnen, Petit et Tixeuil (2004).

Par ailleurs ces deux problèmes (de l'élection et de l'arbre couvrant) ont fait l'objet de nombreuses études dans le domaine du confinement de fautes. Parmi ces travaux, Ghosh et Gupta ont proposé des algorithmes confinant les fautes pour les trois problèmes suivants : l'élection, l'arbre couvrant et le plus court chemin d'abord. Pour le premier problème, Ghosh et Gupta (1996) proposent une version confinant les fautes de l'algorithme d'élection sur un anneau orienté de Lin et Ghosh (1991). Pour le second, ils proposent une version confinant les fautes de l'algorithme de construction d'arbre couvrant dans un système semi-uniforme de Chen et al. (1991). Récemment, Ghosh et He (2000) ont introduit des démons plus forts pour simplifier la conception et la preuve des algorithmes confinant les fautes. Pour cela, ils affectent des priorités aux différentes règles de façon à ce que le processeur fautif puisse exécuter ses règles avant les autres processeurs du système. Ils illustrent leur technique sur l'algorithme de Chen et al. (1991). Pour le troisième problème, Ghosh, Gupta et Pemmaraju (1997) proposent une version confinant les fautes de l'algorithme de construction du plus court chemin d'abord dans un système semi-uniforme de Huang et Chen (1992).

Les travaux précédents sur le problème de l'élection portent sur des systèmes à topologie d'anneau, et ceux sur le problème de l'arbre couvrant se place dans le cadre d'un système semi-uniforme. Nous avons choisi l'algorithme de Afek et al. car il s'applique dans le cadre d'un système non uniforme à topologie quelconque. Ce type de système étant facilement rencontré dans les réseaux à grande échelle, il nous semble souhaitable d'y développer des algorithmes confinant les fautes. Notre contribution consiste donc à introduire la propriété de confinement de fautes dans l'algorithme de Afek, Kutten et Yung.

5.2 Construction d'un arbre couvrant

Dans le cadre des systèmes asynchrones, Afek, Kutten et Yung (1990) ont présenté un algorithme d'élection avec construction d'arbre couvrant dans un modèle à registres partagés. L'idée de l'algorithme consiste à utiliser un contrôle centralisé : un processeur unique, le *leader*, vérifie la consistance du système vis-à-vis de la spécification du problème à résoudre. En cas d'inconsistance, le leader déclenche une ré-initialisation globale du système. La détection locale consiste à vérifier les conditions locales, chacune basée uniquement sur l'état d'un processeur et de ses voisins proches. Le système est dans un état correct si et seulement si des conditions locales sont satisfaites pour tout processeur ; un tel système est dit *localement détectable*. La détection locale déclenche une ré-initialisation globale. Par ailleurs, l'algorithme tient compte des topologies dyna-

miques et permet de circonscrire au voisinage le plus proche l'effet de l'incohérence lié au changement de topologie.

5.2.1 Les hypothèses du modèle

Nous procédons tout d'abord à la définition des variables locales utilisées par chaque processeur. Nous rappelons ensuite le modèle de fonctionnement de l'environnement de l'algorithme. Chaque processeur p_i dispose de deux constantes : son identité, unique parmi les processeurs et que l'on identifiera à i , et l'ensemble de ses voisins, noté $Voisins_i$. Toutes les identités du réseau forment un intervalle fini de \mathbb{N} , noté \mathbb{N}_f . Par conséquent, le diamètre du réseau est connu. De plus, chaque processeur p_i dispose d'autres variables divisées en deux groupes.

1. Les variables relatives à la vérification de la cohérence de l'arbre :
 - $root_i$ variable désignant l'identité de la racine de l'arbre. Elle est à valeur dans \mathbb{N}_f .
 - $dist_i$ variable mesurant la distance de i à la racine. Elle est à valeur dans \mathbb{N}_f .
 - par_i variable désignant le père du processeur p_i dans l'arbre. Si p_i est la racine, alors par_i est égale à i . Sinon, par_i doit être un des voisins. En conséquence, par_i est à valeur dans $Voisins_i \cup \{i\}$.

2. Les variables relatives à la demande de rattachement à un arbre :
 - req_i variable contenant l'identité de l'initiateur de la requête en cours de traitement sur i ; elle prend ses valeurs dans $\mathbb{N}_f \cup \{\perp\}$ où \perp est une valeur spéciale appelée valeur indéfinie.
 - $from_i$ variable contenant l'identité du processeur par qui est transmise la requête en cours de traitement sur i ; elle prend ses valeurs dans $Voisins_i \cup \{i, \perp\}$.
 - to_i variable contenant l'identité du processeur à qui est adressée la requête en cours de traitement sur i (normalement ce doit être par_i) ; elle prend ses valeurs dans $Voisins_i \cup \{i, \perp\}$.
 - dir_i variable indiquant l'état de la requête, en attente d'être acquittée (*ask*) ou acquittée (*grant*) ; elle prend ses valeurs dans $\{ask, grant, \perp\}$.

Les variables de chaque processeur ne sont modifiables que par l'exécution d'une règle de ce processeur. Toute règle instanciée (i.e., dont les paramètres ont une valeur donnée) dont la précondition est vérifiée est dite *applicable*. On appellera *post-condition* d'une règle instanciée la condition induite par l'état des variables modifiées par l'action de la règle.

Toute règle instanciée dont la précondition est vérifiée et dont la post-condition n'est pas vérifiée est dite *exécutable*. L'exécution d'une règle est atomique. En conséquence, une exécution de l'algorithme est une séquence maximale finie ou infinie $\sigma = s_0 a_0 s_1 a_1 \dots s_n a_n \dots$ où s_n est un état de l'algorithme, a_n est une règle instanciée exécutable dans s_n et s_{n+1} l'état résultant de l'exécution de a_n . La valeur d'une variable v dans l'état s_n est notée v^n .

Toute séquence d'exécution doit respecter la contrainte d'équité : si une règle instanciée a sa précondition infiniment souvent vérifiée dans les états de l'exécution alors sa post-condition est aussi infiniment souvent vérifiée. Une séquence d'exécution se divise en rondes successives, chaque ronde étant une sous-suite de la suite des a_n définie de la façon suivante :

- La première ronde débute par l'état s_0 . La $n + 1^{\text{ème}}$ ronde débute par l'état qui termine la $n^{\text{ème}}$ ronde.
- La suite d'actions qui est contenue dans une ronde est la plus petite suite démarrant dans l'état de début de la ronde telle que si I_0 désigne l'ensemble des sites qui n'exécuteront plus d'action à partir de cet état et I_1 le sous-ensemble des sites n'appartenant pas à I_0 et n'ayant pas exécuté d'actions dans cette suite alors pour tout site de I_1 aucune action d'un tel site n'était possible dans aucun des états où l'une des actions de cette suite a été exécutée.

Il est relativement immédiat de vérifier que la notion de ronde est bien définie. L'intérêt de celle-ci est de mesurer la complexité en environnement asynchrone à l'aide du nombre maximal de rondes d'une exécution.

5.2.2 Algorithme initial

L'arbre couvrant est défini par une unique variable par_i pour tous les processeurs p_i du système. Nous appellerons dans la suite par_i , $root_i$ et $dist_i$, les *variables d'arbre* puisqu'elles permettent de vérifier la cohérence de l'arbre grâce aux prédicats suivants :

$$\begin{aligned}
 Root(i) &\equiv root_i = i \wedge dist_i = 0 \\
 Child(i, j) &\equiv root_i = root_j > i \wedge par_i = j \in Neigh_i \wedge dist_i = dist_j + 1 \\
 Tree(i) &\equiv root(i) \vee \exists j : Child(i, j) \\
 Lmax(i) &\equiv \forall j \in Neigh_i : root_p \geq root_j \\
 Sat(i) &\equiv Tree(i) \wedge Lmax(i)
 \end{aligned}$$

Pour se rattacher à un arbre, un processeur envoie une requête à un voisin en utilisant les autres variables, que nous appelons dans la suite *variables de requête*. Ces variables permettent de définir les conditions de rattachement suivantes :

$$\begin{aligned}
 Idle(i) &\equiv req_i = from_i = to_i = dir_i = \perp \\
 Asks(i, j) &\equiv ((root(i) \wedge req_i = i) \vee Child(i, j)) \wedge to_i = j \wedge dir_i = ask \\
 Forw(i, j) &\equiv req_i = req_j \wedge from_i = j \wedge to_j = i \wedge to_i = par_i \\
 Grant(i, j) &\equiv Forw(i, j) \wedge dir_i = grant
 \end{aligned}$$

Une description formelle de l'algorithme est donné pour le modèle à état "read-all" (Algorithme 3). Pour assurer l'auto-stabilisation dans la construction de l'arbre couvrant, tous les processeurs doivent vérifier la condition locale définie par, $\forall i : Sat(i)$. Un prédicat $Sat(i)$ à *faux* induit l'incohérence du voisinage du processeur p_i , qui déclenche alors une ré-initialisation globale du système. Pour cela, le processeur p_i se déconnecte en devenant racine de l'arbre réduit à lui-même (i.e., action de la règle \mathbf{B}_i) et cherche à joindre un autre arbre du réseau à travers l'un de ses voisins. Pour cela, il choisit le voisin p_j rattaché à l'arbre dont l'identité de la racine est la plus grande. Le rattachement se déroule en deux étapes. Tout d'abord, p_i formule sa demande de rattachement (i.e., action de la règle \mathbf{A}_i). Ensuite, il se rattache à l'arbre dès lors qu'il reçoit l'accord de p_j (i.e., action de la règle \mathbf{J}_i). Les quatres actions restantes (i.e., les actions des règles \mathbf{C}_i , \mathbf{F}_i , \mathbf{G}_i et \mathbf{R}_i) décrivent le mécanisme de transmission des requêtes de rattachement initiées uniquement par des processeurs p_i dont le voisinage est cohérent (i.e., $Sat(i)$ est vrai).

Algorithme 3 *Algorithme AKY pour la construction d'un arbre couvrant*

B_i

Précondition : $\neg Tree(i)$

Action : $root_i = par_i = i; dist_i = 0; req_i = from_i = to_i = dir_i = \perp;$

A_i

Précondition : $Tree(i) \wedge \neg Lmax(i)$

Action : Choisir $j \in Voisins_i$ dont la racine $root_j$ est maximale
 $req_i = i; from_i = i; to_i = j; dir_i = ask;$

J_i

Précondition : $Tree(i) \wedge \neg Lmax(i) \wedge Grant(to_i, i)$

Action : $root_i = root_j; par_i = j; dist_i = dist_j + 1; req_i = from_i = to_i = dir_i = \perp;$

C_i

Précondition : $Sat(i) \wedge \neg \exists j : Forw(i, j) \wedge \neg Idle(i)$

Action : $req_i = from_i = to_i = dir_i = \perp;$

F_i

Précondition : $Sat(i) \wedge Idle(i) \wedge Asks(j, i)$

Action : $req_i = req_j; from_i = j; to_i = par_i; dir_i = ask;$

G_iPrécondition : $Sat(i) \wedge Root(i) \wedge Forw(i, j) \wedge dir_i = ask$ Action : $dir_i = grant;$ **R_i**Precon : $Sat(i) \wedge Grants(par_i, i) \wedge dir_i = ask$ Action : $dir_i = grant;$ **fin**

5.3 Confinement de fautes dans la construction d'un arbre couvrant

Dans cette section, nous nous intéressons à la transformation de l'algorithme précédent en un algorithme confinant les fautes de construction d'arbre couvrant. Nous précisons une hypothèse nécessaire pour à la mise en place de l'algorithme : un processeur doit pouvoir lire les variables *des voisins de ses propres voisins*. Ainsi, les variables des voisins d'un processeur et celles des voisins de ses propres voisins peuvent apparaître dans la garde d'une de ses règles. A partir des différentes variables d'un processeur p_i , nous introduisons les abréviations suivantes, utiles dans la suite :

- L'ensemble des sites qui ont le processeur p_i pour père :

$$\mathbf{Fils(i)} \equiv \{k \mid k \in Voisins_i \wedge par_k = i\}$$

- La cohérence locale de la structure d'arbre de p_i

$$\mathbf{Tree(i)} \equiv (root_i = i \wedge dist_i = 0 \wedge par_i = i) \vee \\ (par_i \neq i \wedge root_i = root_{par_i} > i \wedge dist_i = dist_{par_i} + 1)$$

- Le propriétaire de la plus grande racine parmi p_i et ses voisins (en cas d'égalité, i d'abord puis les voisins triés par ordre croissant des identités)

$$\mathbf{ownrmax(i)} \equiv i \quad \text{si } \forall k \in Voisins_i \quad root_i \geq root_k$$

“le plus petit $j \in Voisins_i \mid \forall k \in Voisins_i, root_j \geq root_k$ ” sinon

- La cohérence de voisinage de la structure d'arbre de p_i

$$\mathbf{Sat(i)} \equiv Tree(i) \wedge ownrmax(i) = i$$

- L'absence de traitement de requête en cours sur i

$$\mathbf{Idle(i)} \equiv req_i = \perp \wedge from_i = \perp \wedge to_i = \perp \wedge dir_i = \perp$$

- La requête de rattachement de p_i adressée à p_j

$$\mathbf{Asks(i,j)} \equiv j \in Voisins_i \wedge req_i = i \wedge from_i = i \wedge to_i = j \wedge \\ dir_i = ask \wedge Sat(j) \wedge root_i = i < root_j \wedge par_i = j \wedge \\ \forall k \in Fils_i, root_k = root_j \Rightarrow dist_k = dist_j + 2$$

- La transmission d'une requête non encore acquittée de p_i vers p_j
ForwardsNa(i,j) $\equiv j \in Voisins_i \cup \{i\} \wedge to_i = j \wedge par_i = j \wedge dir_i = ask \wedge Sat(i) \wedge$
 $from_i \neq i \wedge req_i \neq i \wedge req_i \neq \perp \wedge req_i = req_{from_i} \wedge$
 $\forall k \in Fils_i, (root_k = root_i \Rightarrow dist_k = dist_i + 1)$
- La transmission d'une requête acquittée de p_i vers p_j
ForwardsA(i,j) $\equiv j \in Voisins_i \cup \{i\} \wedge to_i = j \wedge par_i = j \wedge dir_i = grant \wedge$
 $Sat(i) \wedge from_i \neq i \wedge req_i \neq i \wedge req_i \neq \perp \wedge req_i = req_{from_i} \wedge$
 $\forall k \in Fils_i, root_k = root_i \Rightarrow dist_k = dist_i + 1$
- La cohérence des variables de requête sur i
Rgcorrect(i) $\equiv Idle(i) \vee \exists j | ForwardsNa(i,j) \vee$
 $(\exists j | ForwardsA(i,j) \wedge dir_{from_i} = ask)$
- L'acquiescement d'une requête venant de p_j par p_i
Grants(i,j) $\equiv from_i = j \wedge dir_i = grant$

5.3.1 Description informelle

L'objectif de l'algorithme recherché est la construction d'un arbre couvrant dont la racine sera le site de plus grande identité. Avant de décrire l'algorithme, nous introduisons des définitions caractérisant les comportements attendus d'un tel algorithme. Précisons tout d'abord, que pour tout processeur p_i , la variables par_i constitue la partie primaire de son état, alors que les variables $root_i$, $dist_i$ et les variables de requête constituent la partie secondaire de son état. D'après la définition 2.30, nous introduisons la notion d'état simplement correct et d'état correct.

Définition 5.1 *Soit un algorithme dont les variables sont données ci-dessus.*

- Une exécution atteint un état simplement correct si à partir de cet état les variables par_i restent inchangées et définissent un arbre couvrant dont la racine est le site de plus grande identité.
- Une exécution atteint un état correct si cet état est simplement correct, $\forall i Idle(i)$ est vérifié, $dist_i$ correspond à la distance effective à la racine, et $root_i$ est positionnée à l'identité de la racine.

De plus, une exécution atteint un état *stable* si et seulement si l'exécution est finie et cet état est le dernier état de l'exécution. A ce titre, un algorithme dont les variables sont données ci-dessus est auto-stabilisant si et seulement si toute exécution est finie et son dernier état est correct.

Définition 5.2 *Soit un algorithme dont les variables sont données ci-dessus.*

- Un état à faute unique est un état issu d'un état correct par un changement quelconque des variables d'un unique site.

- L'algorithme "*x, y*-maîtrise" les fautes uniques si et seulement si toute exécution issue d'un état à faute unique modifie la variable par_i d'au plus un unique site, atteint un état simplement correct après au plus x exécutions de règles, et se termine après au plus y exécutions de règles dans un état correct.

Nous avons conçu un algorithme " $1, \theta(n)$ -maîtrise" que nous décrivons informellement à travers les différentes règles applicables à un processeur p_i :

- $\mathbf{O}_i(j)$: la règle qu'exécute le processeur p_i lorsqu'il estime qu'il est seul fautif, qu'il n'est pas racine de l'arbre couvrant et que p_j devrait être son père (cf. le traitement d'une faute unique décrit dans la section 5.4).
- \mathbf{B}_i : la règle qu'exécute le processeur p_i lorsqu'il détecte que sa structure locale d'arbre est incohérente. De plus, cette règle est applicable uniquement si d'une part la règle \mathbf{O}_i n'est pas applicable et si d'autre part p_i n'estime pas que par_i ($\neq i$) est le seul processeur fautif du réseau. Son action consiste à se déclarer racine de l'arbre et à ré-initialiser les variables de requête.
- \mathbf{A}_i : la règle qu'exécute un processeur p_i dont la structure d'arbre est localement cohérente et qui trouve un voisin dont la racine a une identité supérieure à sa propre racine. Pour qu'elle soit applicable, il faut de plus que p_i n'estime pas que ce voisin soit le seul processeur fautif du réseau. Il définit une requête en attente d'acquiescement, dont il est l'initiateur et qui est adressée à ce voisin. De plus, pour que cette règle soit applicable, il faut que les processeurs déclarant être les fils de p_i pour cette nouvelle racine soient à la bonne distance.
- \mathbf{J}_i : la règle qu'exécute le processeur p_i lorsque sa requête est acquiescée.
- \mathbf{C}_i : la règle qu'exécute le processeur p_i lorsque ses variables de requête ne correspondent pas à une requête en cours ou que celle-ci est acquiescée et que cet acquiescement a été répercuté par celui qui a adressé à p_i cette requête. Pour que cette règle soit applicable, il faut que la structure d'arbre de voisinage soit correcte. Cette condition s'applique aussi aux règles qui suivent.
- $\mathbf{F}_i(j)$: la règle qu'exécute le processeur p_i lorsque ses variables de requête sont indéfinies et qu'une requête d'un de ses fils, p_j , lui est adressée. Elle consiste à modifier les variables de requêtes en conséquence.
- \mathbf{G}_i : la règle qu'exécute le processeur p_i lorsqu'il est racine et que ses variables de requête définissent une requête en attente d'acquiescement. Elle consiste simplement à positionner dir_i à *grant*. Il faut de plus que les processeurs qui déclarent être ses fils avec p_i comme racine soient à bonne distance.

- \mathbf{R}_i : la règle qu'exécute le processeur p_i lorsque ses variables de requête définissent une requête en attente d'acquiescement et que son père a acquiescé sa requête. Elle consiste simplement à positionner dir_i à *grant*. Il faut de plus que les processeurs qui déclarent être ses fils avec la même racine que p_i soient à bonne distance.

Dans la suite, nous précisons comment l'algorithme gère la présence d'une unique faute. Si le processeur fautif est la racine, alors nécessairement la règle \mathbf{B}_i sera exécutable et son exécution stabilisera l'algorithme.

Si le processeur fautif n'est pas la racine, il faut déjà reconnaître la situation ou tout au moins en avoir une estimation vraisemblable. L'estimation repose sur le prédicat $Onelocalfault(i, d, j)$ dont le premier paramètre est le processeur qui évalue le prédicat, le deuxième paramètre est la distance supposée être correcte et le troisième paramètre est un voisin supposé être le père et donc à distance $d - 1$. Il faut tout de suite noter que si $Fils_i$ est vide alors plusieurs couples (d, j) sont possibles et si $Fils_i$ est non vide, une seule valeur d est possible mais plusieurs valeurs de j sont possibles.

Maintenant, nous énumérons les conditions à remplir pour que l'estimation soit vraie. Tous les voisins de p_i doivent avoir même racine r qui doit être strictement supérieure à i et supérieure ou égale aux identités de tous les voisins de p_i . Tous les voisins de p_i différents du processeur racine p_r doivent avoir un père différent d'eux-mêmes et une distance strictement positive. Si ce père est aussi un voisin, alors les distances entre ces deux voisins doivent être cohérentes. Si p_r fait partie des voisins alors sa distance doit être nulle et il doit être son propre père. La présence d'une faute sur p_i doit impliquer que soit la valeur de sa racine n'est pas égale à r , soit la valeur de sa distance n'est pas cohérente avec celle de son père, soit elle n'est pas cohérente avec celle d'un de ses fils. Tous ses fils doivent avoir une valeur de distance $d + 1$ et p_j doit avoir une distance égale à $d - 1$. Enfin si p_i a comme valeur de sa racine r et qu'il se trouve à une distance cohérente avec celle de son père, il se peut que la faute provienne d'un fils. Ceci est exclu si le processeur p_i a plus d'un fils, ou que son unique fils a lui-même un fils à bonne distance (c'est le seul cas où l'estimation porte sur les variables d'un voisin, lui-même voisin du processeur p_i).

Il reste encore à éviter qu'un voisin du processeur fautif effectue une requête de rattachement vers celui-ci s'il a une identité dans sa variable racine (inexistante) de plus grande identité que l'identité du processeur racine de l'arbre.

$$Equalroot(i) \equiv \exists r \geq i : \forall k \in Voisins_i, root_k = r$$

$$Coherent(i) \equiv Tree(i) \wedge \forall k \in Fils_i, dist_k = dist_i + 1 \wedge \forall k \in Voisins_i, root_k = root_i$$

$$Incoherent(i) \equiv \neg Coherent(i)$$

$$Onelocalfault(i, d, j) \equiv j \in Voisins_i \wedge \exists r > i :$$

1. *Incoherent*(i) (*une faute sur le processeur p_i *)
2. $\forall k \in \text{Voisins}_i$: (*voisinage cohérent*)
 - $\text{root}_k = r \wedge k \leq r \wedge (k = r \Rightarrow \text{dist}_r = 0 \wedge \text{par}_r = r)$
 - $k \neq r \Rightarrow \text{par}_k \neq k \wedge \text{dist}_k > 0$
3. $\exists k : \text{Voisins}_i = \{k\} \Rightarrow$ (*cas d'un unique voisin*)
 - $\forall k' \in \text{Voisins}_k \setminus \{i\} : \text{root}_{k'} = r \wedge k' \leq r \wedge (k' = r \Rightarrow \text{dist}_r = 0 \wedge \text{par}_r = r)$
 - $(k \neq r \wedge \text{par}_k \neq i) \Rightarrow \text{dist}_k = \text{dist}_{\text{par}_k} + 1$
 - $\forall k' \neq i \in \text{Fils}_k : \text{dist}_{k'} = \text{dist}_k + 1$
4. (*possibilité d'une correction à l'aide de p_j *)

$$\forall k \in \text{Fils}_i : \text{dist}_k = d + 1 \wedge \text{dist}_j = d - 1$$
5. (*élimination d'une fausse détection due à une unique faute d'un fils*)

$$\text{root}_i = r \wedge \text{dist}_i = \text{dist}_{\text{par}_i} + 1 \wedge \text{Idle}(i) \Rightarrow$$
 - $|\text{Fils}_i| > 1 \vee$
 - $\exists k : \text{Fils}_i = \{k\} \wedge \exists k' \neq i \in \text{Fils}_k \wedge (\forall k' \neq i \in \text{Fils}_k : \text{dist}_{k'} = \text{dist}_k + 1 \vee \text{par}_i = k)$
6. (*élimination d'une fausse détection due à une unique faute du père*)

$$(\text{root}_i = r \wedge \forall k \in \text{Fils}_i : \text{dist}_k = \text{dist}_i + 1 \wedge \text{Idle}(i) \wedge \text{par}_{\text{par}_i} \neq i) \Rightarrow$$

$$(\text{dist}_{\text{par}_i} = \text{dist}_{\text{par}_{\text{par}_i}} + 1 \wedge \forall k' \neq i \in \text{Fils}_{\text{par}_i} : \text{dist}_{k'} = \text{dist}_{\text{par}_i} + 1)$$
7. (*élimination des pères croisés*)

$$(\text{root}_i = r \wedge \text{Idle}(i) \wedge \text{par}_{\text{par}_i} = i \wedge \text{par}_i \neq i) \Rightarrow \forall k \in \text{Fils}_{\text{par}_i} \setminus \{i\} : \text{dist}_k = \text{dist}_{\text{par}_i} + 1$$

Si *Onelocalfault*(i, d, j), la règle $\mathbf{O}_i(j)$ consiste alors à adresser une requête processeur p_j . Cependant, le point-clef consiste à laisser inchangée la distance et à positionner la valeur de la racine à i . Comme cette règle peut être appliquée avec différents processeurs p_j , on impose comme contrainte supplémentaire que dans l'état courant une telle requête n'est pas déjà présente à l'aide du prédicat $\forall k \text{ Not Asks}(i, k)$.

Cependant lors de la présence d'une faute, il faut empêcher les fils du processeur fautif de se détacher de l'arbre par l'exécution de la règle \mathbf{B}_i . Ce qui conduit à introduire le prédicat *Onelocalparfault*(i, d, j) testé par le processeur p_i pour vérifier si son père estime être victime d'une faute unique (d et j sont relatifs à par_i) dans un arbre dont il n'est pas racine.

Onelocalparfault(i, d, j) $\equiv \text{par}_i \neq i \wedge j \in \text{Voisins}_{\text{par}_i} \wedge \exists r > \text{par}_i :$

1. *Incoherent*(par_i) (*une faute sur par_i *)

2. $\forall k \in \text{Voisins}_{par_i} : (*\text{voisinage cohérent}*)$
 - $root_k = r \wedge k \leq r \wedge (k = r \Rightarrow dist_r = 0 \wedge par_r = r)$
 - $k \neq r \Rightarrow par_k \neq k \wedge dist_k > 0$
3. $\text{Voisins}_{par_i} = \{i\} \Rightarrow (*\text{cas d'un unique voisin}*)$
 - $\forall k' \in \text{Voisins}_i \setminus \{par_i\} : root_{k'} = r \wedge k' \leq r \wedge (k' = r \Rightarrow dist_r = 0 \wedge par_r = r)$
 - $\forall k' \neq par_i \in \text{Fils}_i : dist_{k'} = dist_i + 1$
4. $(*\text{possibilité d'une correction à l'aide de } p_j*)$
 $\forall k \in \text{Fils}_{par_i} : dist_k = d + 1 \wedge dist_j = d - 1$
5. $(*\text{élimination d'une fausse détection due à une unique faute d'un fils}*)$
 $(root_{par_i} = r \wedge dist_{par_i} = dist_{par_{par_i}} + 1 \wedge Idle(par_i)) \Rightarrow$
 - $|\text{Fils}_{par_i}| > 1 \vee$
 - $\text{Fils}_{par_i} = \{i\} \wedge \exists k' \neq par_i \in \text{Fils}_i \wedge (\forall k' \neq par_i \in \text{Fils}_i : dist_{k'} = dist_i + 1 \vee par_{par_i} = i)$
6. $(*\text{élimination des pères croisés}*)$
 $(root_{par_i} = r \wedge Idle(par_i) \wedge par_{par_i} = i \wedge par_i \neq i) \Rightarrow \forall k \in \text{Fils}_i \setminus \{par_i\} :$
 $dist_k = dist_i + 1$

Il faut un prédicat analogue pour le cas où par_i est le processeur racine victime d'une faute unique.

$$\begin{aligned}
 \text{Onerootparfault}(i) &\equiv par_i \neq i \wedge \\
 &\quad \forall k \in \text{Voisins}_{par_i} : k < par_i \wedge root_k = par_i \wedge \\
 &\quad \forall k \in \text{Voisins}_i \setminus \{par_i\} : k < par_i \wedge root_k = par_i \wedge \\
 &\quad \forall k \in \text{Fils}_{par_i} : dist_k = 1
 \end{aligned}$$

Un dernier prédicat combine les deux précédents pour éviter un détachement du processeur p_i en cas d'une faute unique du processeur père dont l'identité contenue dans par_i est différente de i .

$$\text{Oneparfault}(i) \equiv \text{Onerootparfault}(i) \vee \exists(d, j) : \text{Onelocalparfault}(i, d, j)$$

Lemme 5.1 Soit p_i un processeur tel que $par_i \neq i$, $par_i \neq root_i$ et $\neg \text{Tree}(i)$ alors $\text{Oneparfault}(i) \Leftrightarrow \exists(d, j) \text{Onelocalparfault}(par_i, d, j)$

Preuve. Des conditions sur par_i , on déduit que $\exists(d, j) \text{Onelocalparfault}(i, d, j)$. Or les conditions de ce prédicat sont identiques aux conditions de $\text{Onelocalparfault}(par_i, d, j)$, exceptée la sixième qui est vérifiée puisque l'on a $\text{Oneparfault}(i) \wedge \neg \text{Tree}(i) \Rightarrow dist_i \neq dist_{par_i} + 1 \vee root_i \neq root_{par_i}$. ■

5.3.2 Algorithme

Une description formelle des règles de l'algorithme confinant les fautes pour le problème de l'arbre couvrant est donnée ci-après (Algorithme 4).

Algorithme 4 *Algorithme confinant les fautes pour la construction d'un arbre couvrant*

O_i(j) (*correction d'une unique faute*)

Précondition : $\exists d \text{ Onelocalfault}(i, d, j) \wedge \forall k \neg \text{Asks}(i, k)$

Action : $root_i = i; par_i = j; req_i = i; from_i = i; to_i = j; dir_i = ask;$

B_i (*devenir racine*)

Précondition : $\forall (k, d) \neg \text{Onelocalfault}(i, d, k) \wedge \neg \text{Tree}(i) \wedge \forall k, \neg \text{Asks}(i, k) \wedge \neg \text{Oneparfault}(i)$

Action : $root_i = par_i = i; dist_i = 0; req_i = from_i = to_i = dir_i = \perp;$

A_i (*requête de permission de rattachement*)

Précondition : $\text{Tree}(i) \wedge \text{ownrmax}(i) \neq i \wedge \text{Sat}(\text{ownrmax}(i)) \wedge$

$\forall k \in \text{Fils}_i, (\text{root}_k = \text{root}_{\text{ownrmax}(i)} \Rightarrow \text{dist}_k = \text{dist}_{\text{ownrmax}(i)} + 2)$

Action : $req_i = i; from_i = i; to_i = \text{ownrmax}(i); dir_i = ask; root_i = i; par_i = \text{ownrmax}(i)$

J_i (*rattachement à l'arbre*)

Précondition : $\exists j \text{Asks}(i, j) \wedge \text{Grants}(j, i) \wedge \text{dist}_j + 1 \in \mathbb{N}_f$

Action : $root_i = root_j; par_i = j; dist_i = dist_j + 1; req_i = from_i = to_i = dir_i = \perp;$

C_i (*initialisation des variables de requête*)

Précondition : $\forall (k, d) \neg \text{Onelocalfault}(i, d, j) \wedge \text{Sat}(i) \wedge \neg \text{Rgcorrect}(i)$

Action : $req_i = from_i = to_i = dir_i = \perp;$

F_i(j) (*transmission d'une requête*)

Précondition : $\text{Sat}(i) \wedge \text{Idle}(i) \wedge (\text{Asks}(j, i) \vee \text{ForwardsNa}(j, i))$

Action : $req_i = req_j; from_i = j; to_i = par_i; dir_i = ask;$

G_i (*acquiescement d'une requête*)

Précondition : $\text{Sat}(i) \wedge i = root_i \wedge \text{ForwardsNa}(i, i)$

Action : $dir_i = grant;$

R_i (*transmission d'un acquiescement*)

Précondition : $\text{Sat}(i) \wedge \exists j (\text{ForwardsNa}(i, j) \wedge \text{Grants}(j, i))$

Action : $dir_i = grant;$

fin

5.4 Preuve de la maîtrise d'une faute unique

Supposons que l'algorithme soit dans un état incorrect obtenu à partir d'un état correct par une faute sur le processeur p_i . Seuls le processeur p_i et ses voisins sont susceptibles d'exécuter des règles. Nous caractérisons les différentes possibilités d'action dans l'état fautif par une suite de lemmes.

Lemme 5.2 *Soit un état incorrect obtenu à partir d'un état correct par une faute sur p_i . Alors aucune des règles \mathbf{J}_i , \mathbf{F}_i , \mathbf{G}_i et \mathbf{R}_i n'est applicable. Par ailleurs, soit p_j un voisin quelconque de p_i , alors aucune des règles \mathbf{A}_j , \mathbf{J}_j , \mathbf{C}_j , \mathbf{G}_j et \mathbf{R}_j n'est applicable. De plus, si $\text{par}_j \neq i$ alors \mathbf{B}_j n'est pas applicable et si $\text{par}_i \neq j$ alors \mathbf{O}_j n'est pas applicable.*

Preuve. D'une part, pour le processeur p_i aucune des règles \mathbf{J}_i , \mathbf{F}_i , \mathbf{G}_i et \mathbf{R}_i n'est applicable puisque pour tout processeur p_j voisin de p_i , $\text{Idle}(j)$ est vérifié.

D'autre part, en raison des valeurs des variables de requête (toutes indéfinies \perp), pour tout p_j un voisin quelconque de p_i , aucune des règles \mathbf{J}_j , \mathbf{C}_j , \mathbf{G}_j et \mathbf{R}_j n'est applicable. De plus, la valeur de $\text{ownrmax}(j)$ est égale à j ou à i , selon la valeur de root_i (comme illustré sur la figure 5.1) :

- $r' < r$: dans ce cas, $\text{ownrmax}(j) = j$ et donc $\text{ownrmax}(j) \neq j$ n'est pas vérifié ;
- $r' > r$: dans ce cas, $\text{ownrmax}(j) = i$. De plus, si $\text{par}_i = i$ alors $\text{root}_i = r'$ ne peut être égale à i puisque r' est une identité supérieure à r (la plus grande identité du réseau). Sinon, avec $\text{par}_i \neq i$ nous avons $\text{root}_{\text{par}_i} = r \neq r' = \text{root}_i$. Dans les deux cas, le prédicat $\text{Tree}(\text{ownrmax}(j))$ est faux et par conséquent le prédicat $\text{Sat}(\text{ownrmax}(j))$ est faux.

Dans les deux cas (i.e., $\text{ownrmax}(j)$ est égale à j ou à i), la règle \mathbf{A}_j n'est pas applicable.

Par ailleurs, supposons que le processeur p_j ne soit pas un fils de p_i , alors deux cas se présentent :

- si p_j a une distance cohérente avec celle de son éventuel père $\text{par}_j \neq i$, alors $\text{Tree}(j)$ est vérifié et \mathbf{B}_j n'est pas applicable.
- si de plus p_j n'est pas le père de p_i , alors deux cas sont possibles. Soit $\text{root}_i \neq r$, et alors la deuxième condition du prédicat $\text{Onelocalfault}(j, d, j')$ n'est pas vérifiée indépendamment de d et de j' . Soit $\text{root}_i = r$, et $\text{Coherent}(j)$ est vérifié. Dans les deux cas, \mathbf{O}_j n'est pas applicable.

■

Corollaire 5.3 *Soit un état incorrect obtenu à partir d'un état correct par une faute sur p_i . Alors les seules règles éventuellement exécutables sont : \mathbf{O}_i , \mathbf{B}_i , \mathbf{A}_i , \mathbf{C}_i , \mathbf{F}_k pour $k \in \text{Voisins}_i$, \mathbf{B}_k pour $k \in \text{Voisins}_i$ tel que $\text{par}_k = i$, et \mathbf{O}_k pour $k \in \text{Voisins}_i$ tel que $\text{par}_i = k$ ou $\text{par}_k = i$.*

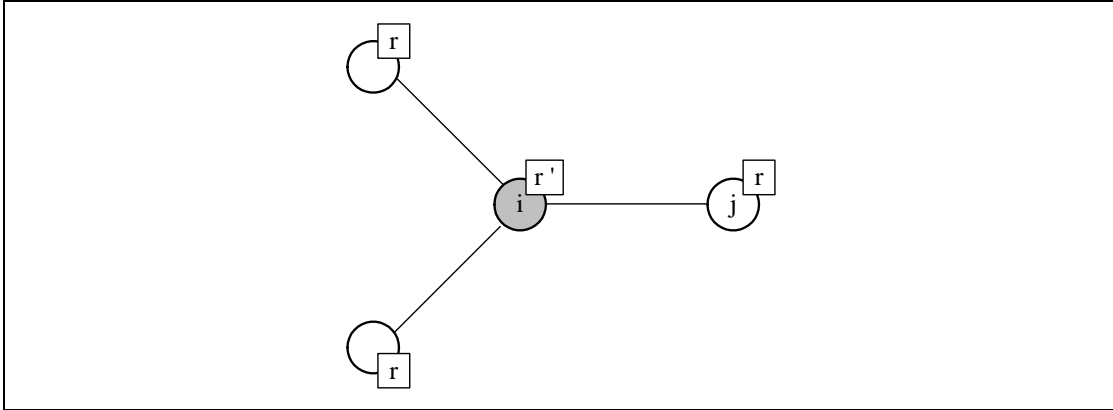


FIG. 5.1 – Un changement de racine

Lemme 5.4 *Soit un état incorrect obtenu à partir d'un état correct par une faute sur p_i . Supposons que dans cet état à faute unique, $Coherent(i)$ soit vérifié, alors la seule règle applicable est C_i et son application conduit à un état correct.*

Preuve. Puisque $Coherent(i)$ est vérifié, tous les sites ont une structure d'arbre correcte (père et fils). Donc le raisonnement du lemme précédent s'étend à i , son père (éventuel) et ses (éventuels) fils, ce qui signifie que les règles O et B ne sont pas applicables à ces sites. De plus, le fait que $Coherent(i)$ soit vrai implique que p_i et tous ses voisins ont la même valeur comme racine, par conséquent $ownrmax(i)$ est égale à i et la règle A_i n'est pas applicable. Pour tout processeur p_j voisin de p_i , la règle F_j n'est pas applicable puisque $Asks(i, j)$ n'est pas satisfait. Enfin, l'une des variables de requête de p_i doit être définie (sinon l'état obtenu serait correct) mais dans ce cas, par un simple examen de la précondition et en utilisant le fait que les variables des autres sites sont indéfinies, on vérifie que C_i est applicable. Son application conduit de manière évidente à un état correct. ■

Lemme 5.5 *Soit un état incorrect obtenu à partir d'un état correct par une faute sur p_i . Supposons que dans cet état, $Coherent(i)$ ne soit pas vérifié et que p_i soit le processeur de plus grande identité, alors la seule règle applicable est B_i et son application conduit à un état correct.*

Preuve. Puisque p_i est le processeur de plus grande identité dans le réseau, alors le prédicat $Onelocal\ fault(i, d, j)$ ne peut jamais être satisfait. Donc O_i n'est pas exécutable. Par contre, $Onerootpar\ fault(k)$ est satisfait pour tout $k \in Fils_i$ et par conséquent B_k n'est pas exécutable. De même pour p_k voisin de p_i , $Onelocal\ fault(k, d, j)$ ne peut être satisfait en raison de la deuxième condition qui ne peut être vérifiée par p_i (puisque l'on a $Incoherent(i)$) et par conséquent $O_k(i)$ n'est pas exécutable. Pour la même raison, $Sat(i)$ n'est pas satisfait et C_i n'est pas exécutable. La règle $F_k(i)$ n'est pas exécutable

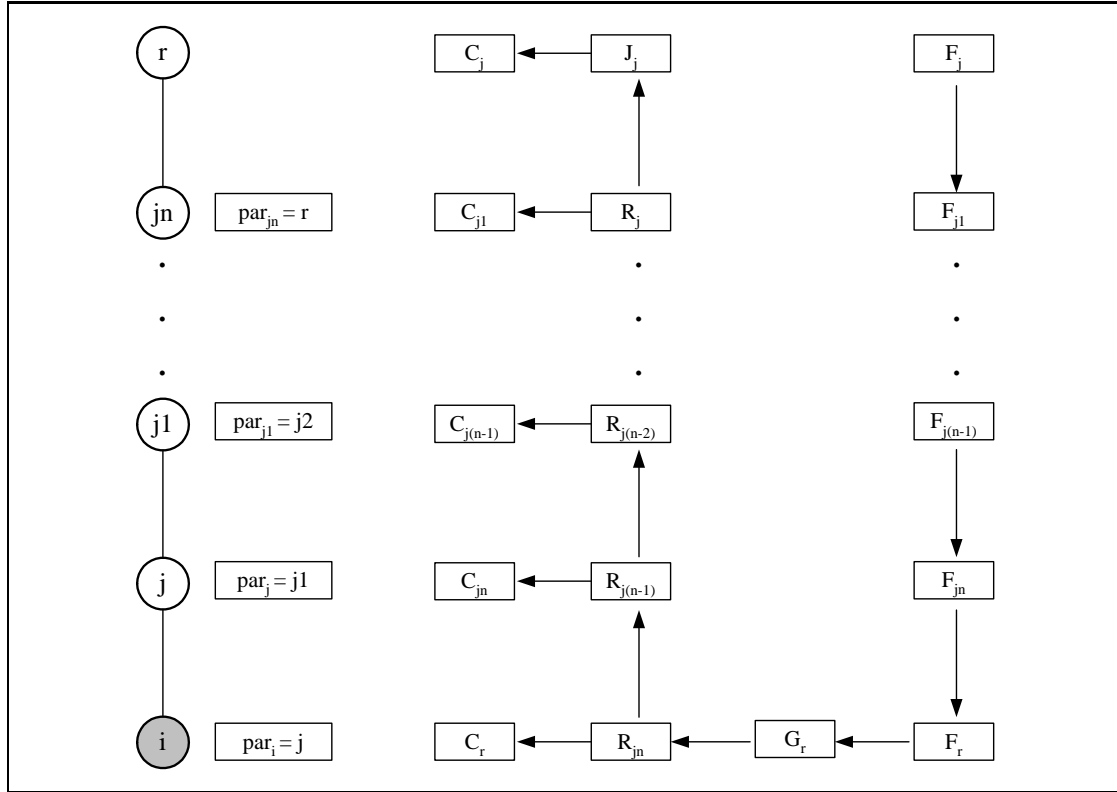


FIG. 5.2 – Un rattachement à partir d'un état incorrect du lemme 5.6

car $Tree(i)$ n'est pas satisfait. Enfin, le prédicat $Coherent(i)$ est faux du fait que $Tree(i)$ est faux, soit

- $\exists k \in \text{Fils}_i$ tel que $dist_k \neq dist_i + 1$. La faute étant sur le processeur racine p_i , on a $dist_i \neq 0$ et par conséquent $Tree(i)$ est faux.
- $\exists k \in \text{Voisins}_i$ tel que $root_k = root_i$. Dans ce cas, $root_i \neq i$ et par conséquent $Tree(i)$ est faux.

En résumé, le fait que $Coherent(i)$ soit faux pour $i = r$ induit dans tous les cas que le prédicat $Tree(i)$ est faux. Par conséquent la règle \mathbf{A}_i n'est pas exécutable.

Examinons la précondition de \mathbf{B}_i : $Onelocalfault(i, d, j)$ ne peut jamais être satisfait, $Tree(i)$ n'est pas satisfait, $Asks(i, k)$ ne peut être satisfait car $i = root_k$, $Oneparfault(i)$ n'est pas satisfait puisque pour p_k voisin de p_i , $Onelocalfault(k, d, j)$ ne peut être satisfait et $Onerootparfault(k)$ n'est pas satisfait en raison de la valeur de i . En conclusion, \mathbf{B}_i est exécutable et son application conduit de manière évidente à un état correct. ■

Lemme 5.6 *Soit un état incorrect obtenu à partir d'un état correct par une faute sur p_i . Supposons que $Coherent(i)$ ne soit pas vérifié et que $i \neq r$ (r étant la plus grande identité dans le réseau). Soit p_j un voisin quelconque de p_i , alors $F_j(i)$ est applicable si*

et seulement si $Asks(i, j)$ est vérifié. Dans ce dernier cas, l'état incorrect est simplement correct et la suite de règles qui s'exécutera jusqu'à l'obtention d'un état correct sera composée de \mathbf{F}_k pour p_k sur le chemin de p_j à p_r , ensuite \mathbf{G}_r et enfin \mathbf{R}_k sur le chemin de p_r à p_j suivie de \mathbf{J}_i ; cette dernière suite d'instructions se trouvant mixée avec les \mathbf{C}_k sur le chemin de p_r à p_j (l'ordre partiel à respecter par le mélange est indiqué sur la figure 5.2).

Preuve. Supposons que la précondition de $\mathbf{F}_j(i)$ soit vérifiée. Dans ce cas, le prédicat $ForwardsNa(i, j)$ ne peut être vérifié puisque si $from_i \neq i$ alors $req_{from_i} = \perp$. Par conséquent, $Asks(i, j)$ est vérifié.

Supposons que $Asks(i, j)$ soit vérifié. Dans ce cas, le processeur p_i ne peut être le père du processeur p_j car cela contredirait la condition sur les distances du prédicat $Asks$. Donc $Tree(j)$ est vérifié et puisque $root_i = i$, alors le prédicat $Sat(j)$ est aussi vérifié. Enfin, $Idle(j)$ est vérifié. Par conséquent $\mathbf{F}_j(i)$ est exécutable.

Puisque $Asks(i, j)$ est vérifié, la structure engendrée par les variables par_k est bien un arbre de couverture ayant pour racine le processeur d'identité r . De plus, par examen des prédicats, $Onelocalfault(i, d, j)$ est vérifié (pour d la distance de p_i à p_r dans l'arbre) et $Oneparlocalfault(k, d, j)$ est vérifié pour tout $k \in Fils_i$. Bien entendu, $Tree(i)$ n'est pas vérifié puisque $par_i \neq i$ et $root_i = i < root_j = root_{par_i}$. Enfin, $Onelocalfault(k, d, k')$ ne peut jamais être vérifié pour $k \in Voisins_i$ car $root_i = i \neq r$. Par conséquent ni \mathbf{O}_i , ni \mathbf{B}_i , ni \mathbf{A}_i , ni \mathbf{C}_i , ni \mathbf{O}_k , ni \mathbf{B}_k pour $k \in Voisins_i$ ne sont applicables. En appliquant la première partie de la preuve on a aussi que \mathbf{F}_k n'est pas applicable pour $k \neq j$ voisin de p_i . En conclusion \mathbf{F}_j est la seule règle applicable.

La structure de la suite de l'exécution se vérifie de manière immédiate à partir des définitions des règles. ■

Lemme 5.7 *Soit un état incorrect obtenu à partir d'un état correct par une faute sur p_i . Supposons que $Coherent(i)$ ne soit pas vérifié, que $i \neq r$ (r étant la plus grande identité dans le réseau) et que pour tout p_j voisin de p_i , $Asks(i, j)$ ne soit pas vérifié.*

1. Si $root_i \neq r$, alors $\exists j$ tel que $\mathbf{O}_i(j)$ soit exécutable, et \mathbf{A}_i est la seule autre règle éventuellement exécutable. L'exécution de l'une de ces deux règles rendra la deuxième non applicable et conduira le système à un état du lemme 5.6.
2. Si $root_i = r$ et $par_i = i$, alors $\exists j$ tel que $\mathbf{O}_i(j)$ soit exécutable, seule la règle \mathbf{O}_i est exécutable et cette règle conduit à un état du lemme 5.6.
3. Si $root_i = r$, $par_i \notin Fils_i \cup \{i\}$ et $dist_i \neq dist_{par_i} + 1$, alors $\exists j$ tel que $\mathbf{O}_i(j)$ soit exécutable, seule la règle \mathbf{O}_i est exécutable et cette règle conduit à un état du lemme 5.6.
4. Si $root_i = r$, $par_i \notin Fils_i \cup \{i\}$ et $dist_i = dist_{par_i} + 1$, alors :
 - (a) Si $|Fils_i| > 1$, alors $\exists j$ tel que $\mathbf{O}_i(j)$ soit exécutable, seule la règle \mathbf{O}_i est exécutable et cette règle conduit à un état du lemme 5.6.

- (b) Si $Fils_i = \{k\}$ et $|Fils_k| = \emptyset$, alors $\exists j$ tel que $\mathbf{O}_k(j)$ soit exécutable, seule la règle \mathbf{O}_k est exécutable et cette règle conduit à un état du lemme 5.6.
- (c) Si $Fils_i = \{k\}$ et $|Fils_k| > 0$, alors \mathbf{O}_i est exécutable et \mathbf{O}_k est la seule autre règle éventuellement exécutable. L'exécution de l'une de ces deux règles rendra la deuxième non applicable et conduira le système à un état du lemme 5.6 vis-à-vis de i ou de k .

5. Si $root_i = r$ et $par_i = j \in Fils_i$, alors :

- (a) Si $dist_j = dist_i + 1$, alors :
 - i. Si $|Fils_j| > 1$, alors $\exists k$ tel que $\mathbf{O}_i(k)$ soit exécutable, seule la règle \mathbf{O}_i est exécutable et cette règle conduit à un état du lemme 5.6.
 - ii. Si $Fils_j = \{i\}$, alors \mathbf{O}_i est exécutable et \mathbf{O}_j est la seule autre règle éventuellement exécutable. L'exécution de l'une de ces deux règles rendra la deuxième non applicable et conduira le système à un état du lemme 5.6 vis-à-vis de i ou de j .
- (b) Si $dist_i = dist_j + 1$ et $dist_j \neq dist_i + 1$, alors :
 - i. Si $|Fils_i| > 1$, alors $\exists k$ tel que $\mathbf{O}_i(k)$ soit exécutable, seule la règle \mathbf{O}_i est exécutable et cette règle conduit à un état du lemme 5.6.
 - ii. Si $Fils_i = \{j\}$, alors \mathbf{O}_i est exécutable et \mathbf{O}_j est la seule autre règle éventuellement exécutable. L'exécution de l'une de ces deux règles rendra la deuxième non applicable et conduira le système à un état du lemme 5.6 vis-à-vis de i ou de j .
- (c) Si $dist_i \neq dist_j + 1$ et $dist_j \neq dist_i + 1$, alors :
 - i. Si $|Fils_i| > 1$, alors $\exists k$ tel que $\mathbf{O}_i(k)$ soit exécutable, seule la règle \mathbf{O}_i est exécutable et cette règle conduit à un état du lemme 5.6.
 - ii. Si $Fils_i = \{j\}$ et $Fils_j = \{i\}$, alors \mathbf{O}_i est exécutable et \mathbf{O}_j est la seule autre règle éventuellement exécutable. L'exécution de l'une de ces deux règles rendra la deuxième non applicable et conduira le système à un état du lemme 5.6 vis-à-vis de i ou de j .
 - iii. Si $Fils_i = \{j\}$ et $|Fils_j| > 1$, alors $\exists k$ tel que $\mathbf{O}_i(k)$ soit exécutable, seule la règle \mathbf{O}_i est exécutable et cette règle conduit à un état du lemme 5.6.

Preuve. Nous remarquons d'abord que toutes les conditions de $Onelocalfault(i, d, j)$ (avec p_j le père de p_i et d la distance à la racine dans l'état correct) sont vérifiées sauf éventuellement la cinquième, la sixième et la septième. De plus pour tout voisin p_k de p_i , $\mathbf{F}_k(i)$ n'est pas applicable car d'une part $Asks(i, k)$ n'est pas vérifié et d'autre part $ForwardsNa(i, k)$ n'est pas vérifié car soit $Sat(i)$ n'est pas vérifié, soit $req_{from_i} = \perp \neq req_i$, si $req_i \neq \perp$ et $from_i \neq i$.

Ensuite, dès lors que la faute sur p_i ne change pas la valeur de sa variable racine r , qui est la même que celles de tous les voisins de p_i , $ownrmax(i)$ est égale à i . Par conséquent, la règle \mathbf{A}_i n'est pas applicable.

1. Supposons que $root_i \neq r$. Dans ce cas, les trois dernières conditions du prédicat $Onelocalfault(i, d, j)$ sont vérifiées, donc $\mathbf{O}_i(j)$ est exécutable et par conséquent ni \mathbf{B}_i , ni \mathbf{C}_i ne sont exécutables. En vertu du lemme 5.1, \mathbf{B}_k n'est pas applicable pour un $k \in Fils_i$. De plus, \mathbf{B}_k n'est pas applicable pour un $k \in Voisins_i \setminus \{Fils_i\}$ car $Tree(k)$ est vérifié. De plus, $root_i \neq r$ implique que la deuxième condition ou la troisième sont fausses et donc $Onelocalfault(k, d, k')$ n'est pas vérifié. Par conséquent \mathbf{O}_k n'est applicable pour aucun voisin de p_i . Enfin, si cette faute sur p_i entraîne que $Tree(i)$ devient faux, alors la règle \mathbf{A}_i n'est pas exécutable. Dans le cas contraire, \mathbf{A}_i est la seule autre règle éventuellement exécutable. L'exécution de l'une de ces deux règles rendra d'une part le prédicat $Asks(i, j)$ vrai et par conséquent la deuxième règle non applicable, et conduira d'autre part le système à un état du lemme 5.6. Un exemple est représenté sur la figure 5.3.
2. Supposons que $root_i = r$ et $par_i = i$. Dans le prédicat $Onelocalfault(i, d, j)$, la cinquième condition est vérifiée car $dist_i \neq dist_i + 1 = dist_{par_i} + 1$, la sixième est vérifiée car $par_{par_i} = i$ et la septième est vérifiée car $par_i = i$. Par conséquent $\mathbf{O}_i(j)$ est exécutable et ni \mathbf{B}_i , ni \mathbf{C}_i ne sont applicables. Pour tout $k \in Voisins_i$, $Onelocalfault(k, d, k')$ n'est pas vérifié car $Coherent(k)$ est vérifié et par conséquent \mathbf{O}_k n'est pas applicable. Enfin, \mathbf{B}_k n'est pas applicable car $Tree(k)$ est vérifié pour tout $k \in Voisins_i$. Donc la seule règle exécutable est \mathbf{O}_i et son exécution pour un processeur j quelconque conduit naturellement à un état du lemme 5.6. Un exemple est représenté sur la figure 5.4.
3. Supposons que $root_i = r$ et $par_i \notin Fils_i \cup \{i\}$ et $dist_i \neq dist_{par_i} + 1$. Dans ce cas, la cinquième condition du prédicat $Onelocalfault(i, d, j)$ est vérifiée car $dist_i \neq dist_{par_i} + 1$. Puisque $par_{par_i} \neq i$, la structure locale d'arbre de par_i est correcte excepté vis-à-vis de son fils p_i , et par conséquent la sixième et la septième condition sont vérifiées. Donc $\mathbf{O}_i(j)$ est exécutable et ni \mathbf{B}_i , ni \mathbf{C}_i ne sont applicables. De plus pour tout $k \in Voisins_i \setminus Fils_i \setminus \{par_i\}$, $Onelocalfault(k, d, k')$ n'est pas vérifié car $Coherent(k)$ est vérifié. Pour par_i , $Onelocalfault(par_i, d, k')$ n'est pas vérifié car la cinquième condition n'est pas vérifiée dans le cas où $|Fils_{par_i}| = \{i\}$, et la quatrième ne l'est pas dans le cas où $|Fils_{par_i}| > 1$. Enfin, pour tout $k \in Fils_i$, $Onelocalfault(k, d, k')$ n'est pas vérifié car la sixième condition n'est pas vérifiée dans le cas où $|Voisins_k| > 1$, et la troisième condition ne l'est pas dans le cas où $Voisins_k = \{i\}$. En vertu du lemme 5.1, \mathbf{B}_k n'est pas applicable pour un $k \in Fils_i$. De plus, \mathbf{B}_k n'est pas applicable pour un $k \in Voisins_i \setminus \{Fils_i\}$ car $Tree(k)$ est vérifié. En outre, $\neg Onelocalfault(k, d, k')$ implique que \mathbf{O}_k n'est applicable pour aucun voisin de p_i . Enfin l'exécution de \mathbf{O}_i pour un j quelconque conduit naturellement à un état du lemme 5.6. Un exemple est représenté sur la figure 5.5.

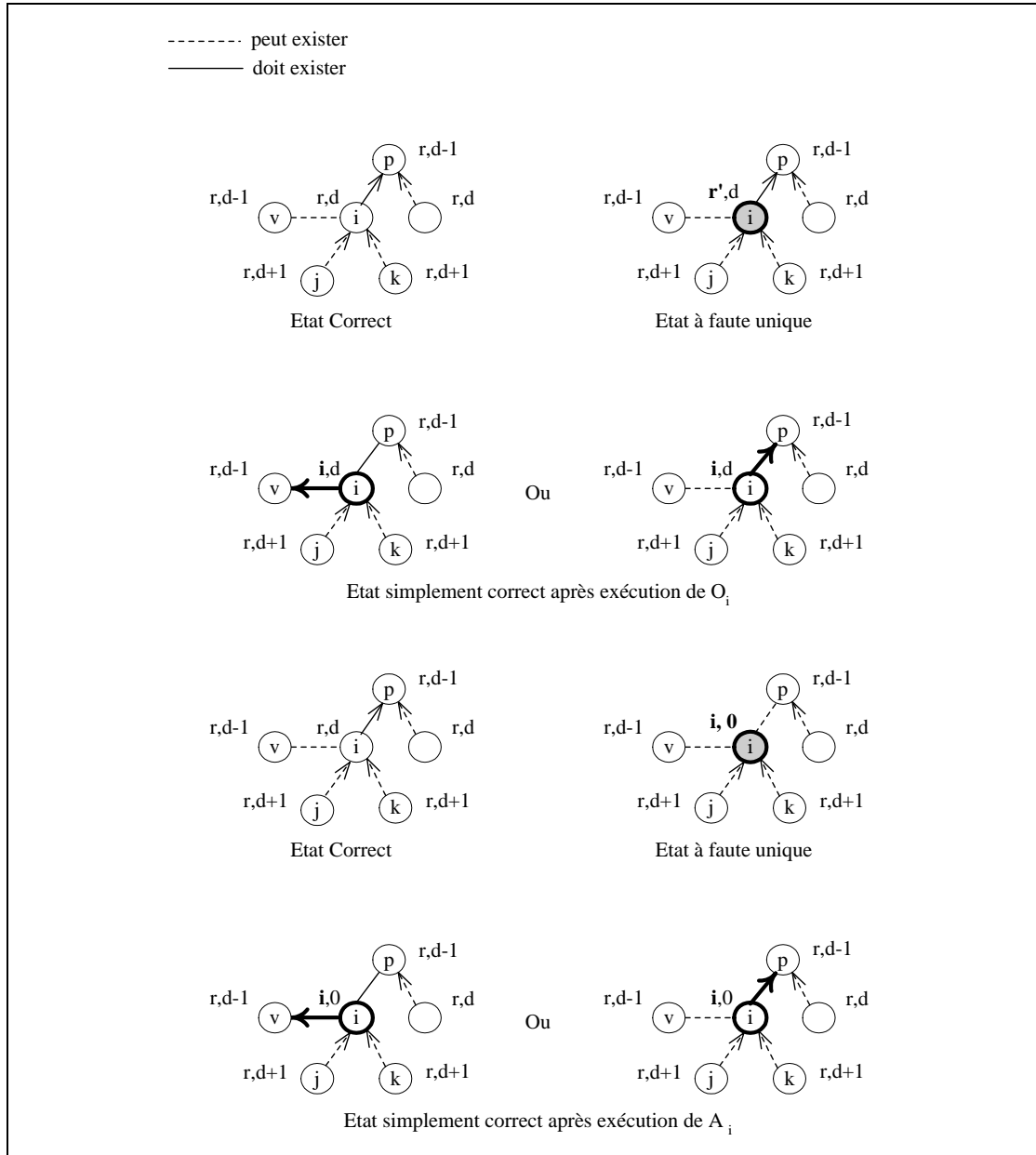


FIG. 5.3 – Un exemple du cas 1

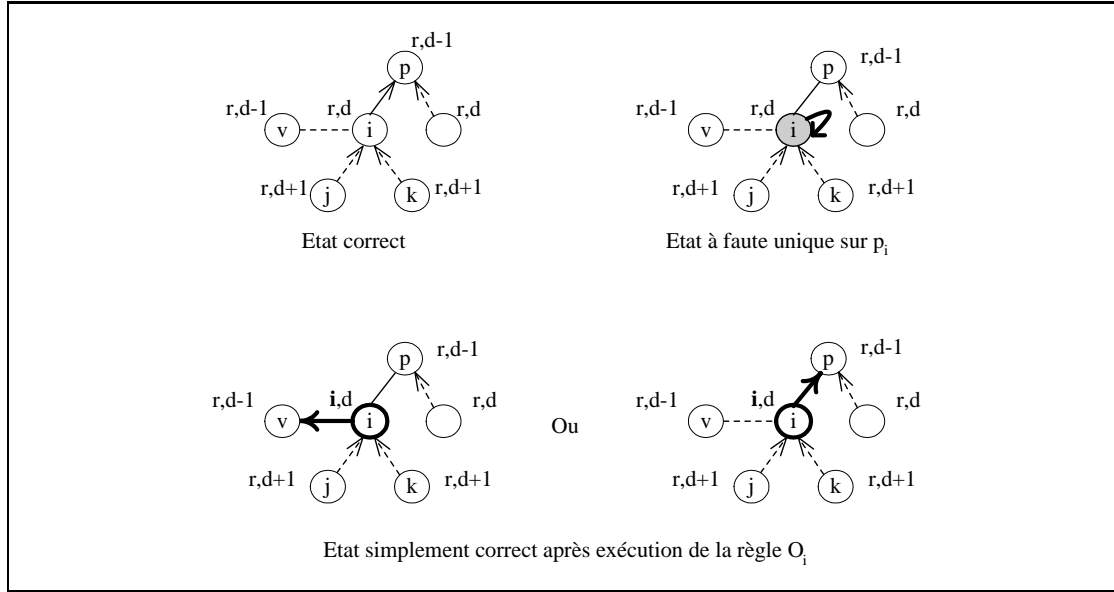


FIG. 5.4 – Un exemple du cas 2

4. Supposons que $root_i = r$, $par_i \notin Fils_i \cup \{i\}$ et $dist_i = dist_{par_i} + 1$. Nécessairement $|Fils_i| > 0$ sinon la structure locale d'arbre de p_i ainsi que l'état du système seraient corrects. Tout d'abord, pour le processeur père de p_i , le prédicat $Onelocal\ fault(par_i, d, j)$ n'est pas vérifié car $Coherent(par_i)$ est vrai. Pour la même raison, $Onelocal\ fault(k, d, j)$ n'est pas vérifié pour tout $k \in Voisins_i \setminus \{Fils_i\} \setminus \{par_i\}$. Par conséquent la règle \mathbf{O}_k n'est pas applicable ainsi que la règle \mathbf{B}_k car $Tree(k)$ est vérifié, pour tout $k \in Voisins_i \setminus \{Fils_i\}$. Par ailleurs, nous distinguons trois cas suivant le nombre de fils de p_i :

(a) $|Fils_i| > 1$: puisque $par_{par_i} \neq i$, la structure locale d'arbre de par_i est correcte excepté vis-à-vis de son fils p_i et par conséquent la sixième et la septième condition sont vérifiées. De plus, la cinquième condition est vérifiée puisque les deux assertions de l'implication sont vraies. Par conséquent, le prédicat $Onelocal\ fault(i, d, j)$ est vérifié, alors que pour tout $k \in Fils_i$, le prédicat $Onelocal\ fault(k, d, k')$ n'est pas vérifié soit parce que la troisième condition ne l'est pas si $Voisins_k = \{i\}$, soit parce que la sixième ne l'est pas dans le cas contraire. Par conséquent, pour tout $k \in Fils_i$, la règle \mathbf{O}_k n'est applicable et la règle \mathbf{B}_k non plus en vertu du lemme 5.1. Ainsi, la règle $\mathbf{O}_i(j)$ est exécutable et par conséquent ni \mathbf{B}_i , ni \mathbf{C}_i ne sont exécutables. La seule règle exécutable est donc \mathbf{O}_i et son exécution conduit à un état du lemme 5.6. Un exemple est représenté sur la figure 5.6.

(b) $Fils_i = \{k\}$ et $|Fils_k| = \emptyset$: dans ce cas, $Onelocal\ fault(i, d, j)$ n'est pas

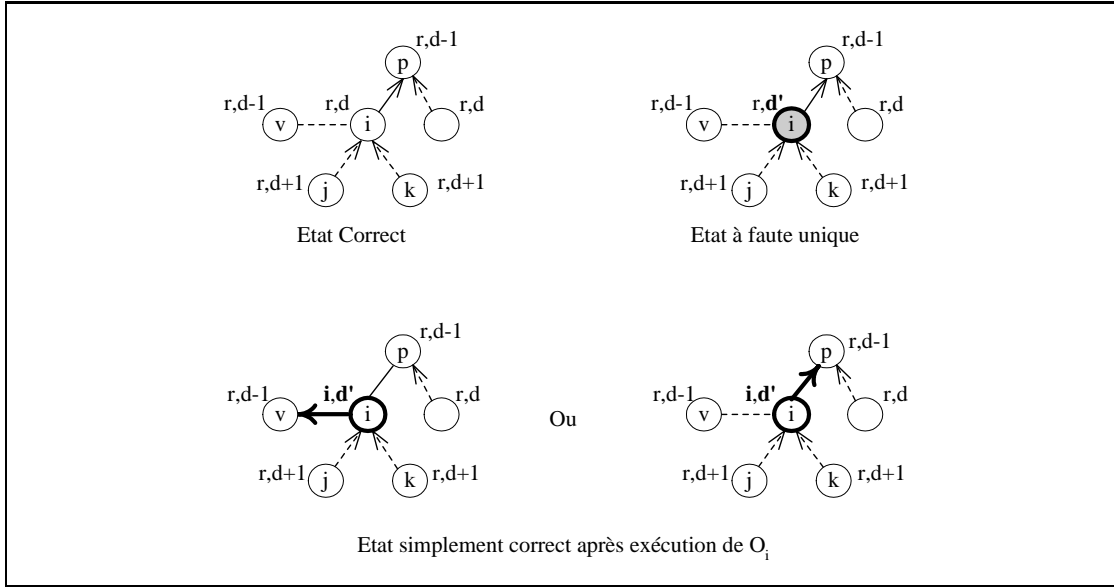


FIG. 5.5 – Un exemple du cas 3

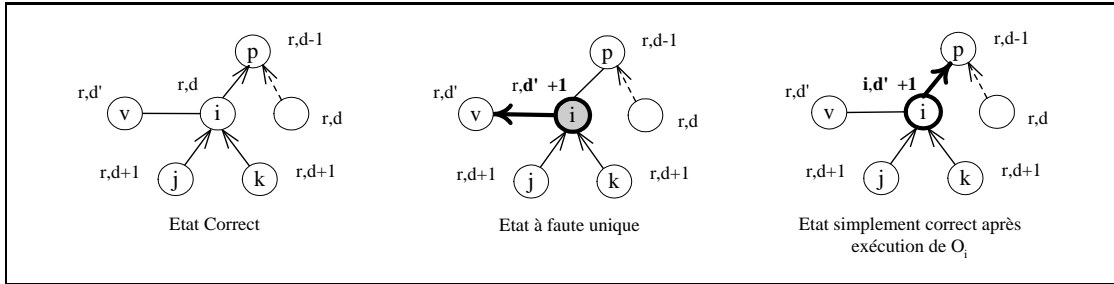


FIG. 5.6 – Un exemple du cas 4a

vérifié car l'assertion à gauche de l'implication, dans la cinquième condition, est vraie et celle qui est à droite est fausse. Par conséquent la règle O_i n'est pas applicable. Il en est de même pour B_i et C_i qui ne sont pas applicables car $Tree(i)$ est vérifié. Pour tout $k \in Fils_i$, le prédicat $Onelocalfault(k, d, k')$ est vérifié et par conséquent la règle O_k est exécutable alors que la règle B_k ne l'est pas. Par conséquent, il existe un j tel que $O_k(j)$ soit la seule règle exécutable et dont l'exécution conduit à un état du lemme 5.6. Un exemple est représenté sur la figure 5.7.

- (c) $Fils_i = \{k\}$ et $|Fils_k| > 0$: dans ce cas, l'assertion à droite de l'implication, dans la cinquième condition du prédicat $Onelocalfault(i, d, j)$, est vraie. Donc ladite condition est vérifiée ainsi que les deux suivantes car $dist_k \neq dist_i + 1$ et $par_{par_i} \neq i$. Par conséquent, $Onelocalfault(i, d, j)$ est vérifié et la règle $O_i(j)$ est exécutable mais les règles B_i et C_i ne le sont pas. De plus, pour

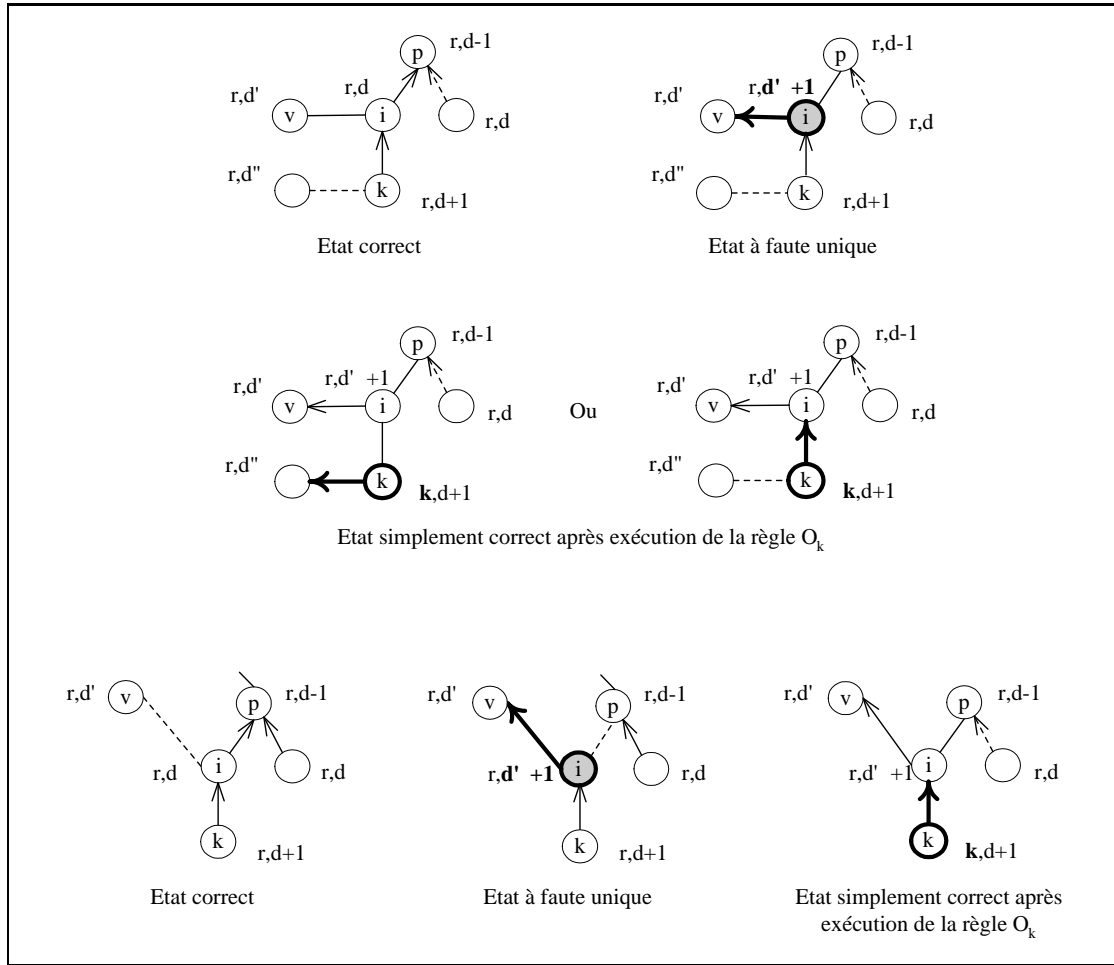


FIG. 5.7 – Un exemple du cas 4b

tout $k \in Fils_i$, le prédicat $Onelocal\ fault(k, d, k')$ est vérifié et par conséquent la règle O_k est la seule autre règle éventuellement exécutable alors que B_k ne l'est pas. L'exécution de l'une de ces deux règles rendra la deuxième non applicable et conduira le système à un état du lemme 5.6 vis-à-vis de i ou de k . Un exemple est représenté sur la figure 5.8.

5. Supposons que $root_i = r$ et $par_i = j \in Fils_i$. Nécessairement $|Fils_i| > 0$ et $|Fils_j| > 0$. Pour tout $k \in Voisins_i \setminus Fils_i$, le prédicat $Onelocal\ fault(k, d, k')$ n'est pas vérifié car $Coherent(k)$ est vérifié et par conséquent les règles B_k et O_k ne sont pas applicables. Nous distinguons trois cas suivant la valeur de $dist_i$ et celle de $dist_j$:

- (a) $dist_j = dist_i + 1$: dans ce cas, le prédicat $Onelocal\ fault(i, d, k)$ est vérifié. En effet, la septième condition est vérifiée, $par_{par_i} = i$ rend la sixième condition vraie et $dist_i \neq dist_{par_i} + 1$ rend la cinquième condition vraie. Par conséquent,

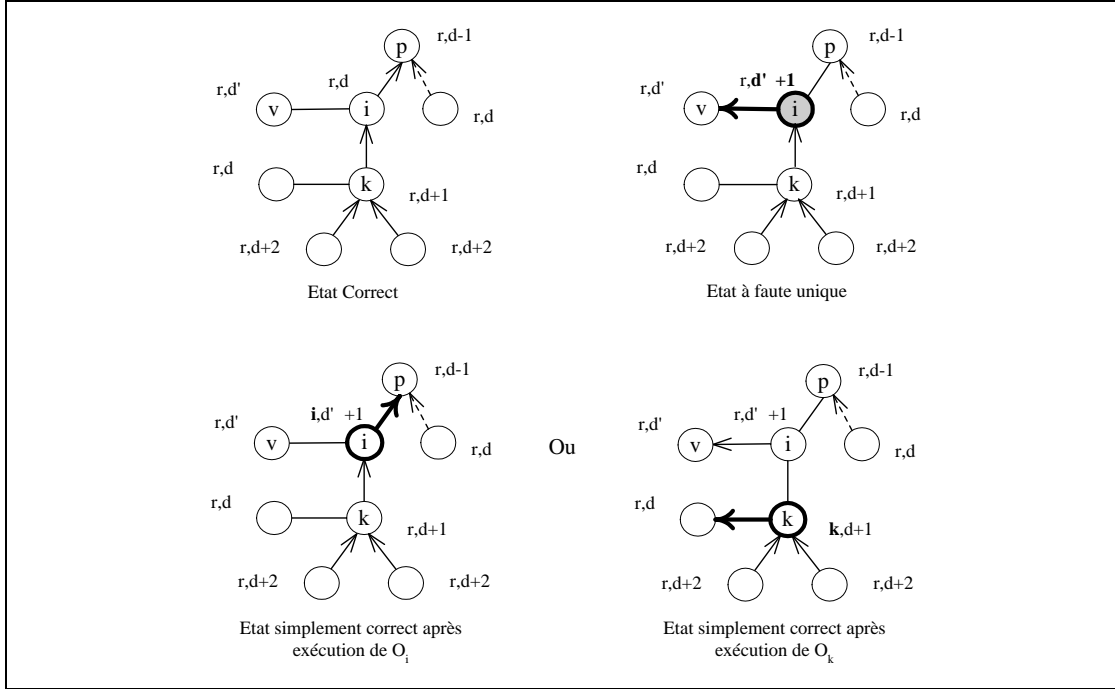


FIG. 5.8 – Un exemple du cas 4c

\mathbf{O}_i est exécutable mais les règles \mathbf{B}_i et \mathbf{C}_i ne le sont pas. De plus, pour tout $k \in \text{Fils}_i \setminus j$, le prédicat $\text{Onelocal fault}(k, d, k')$ n'est pas vérifié car $\text{Coherent}(k)$ est vérifié et par conséquent ni \mathbf{B}_k ni \mathbf{O}_k ne sont applicables. Par ailleurs, nous distinguons deux cas suivant le nombre de fils de p_j :

- i. $|\text{Fils}_j| > 1$: dans ce cas, $\text{Onelocal fault}(j, d, k')$ n'est pas vérifié puisque la quatrième condition ne l'est pas. Ainsi, nous nous trouvons dans un cas analogue à celui décrit dans 4(a). A savoir, il existe k tel que la règle $\mathbf{O}_i(k)$ soit exécutable et son exécution conduit à un état du lemme 5.6. Un exemple est représenté sur la figure 5.9.
- ii. $\text{Fils}_j = \{i\}$: dans ce cas, $\text{Onelocal fault}(j, d, k')$ est vérifié et nous retrouvons le même résultat que celui du cas 4(c). A savoir, \mathbf{O}_i est exécutable et \mathbf{O}_j est la seule autre règle éventuellement exécutable. L'exécution de l'une de ces deux règles rendra la deuxième non applicable et conduira le système à un état du lemme 5.6 vis-à-vis de i ou de j . Un exemple est représenté sur la figure 5.9.

- (b) $\text{dist}_i = \text{dist}_j + 1$ et $\text{dist}_j \neq \text{dist}_i + 1$: pour tout $k \in \text{Fils}_i \setminus \{j\}$, dist_k est différente de $\text{dist}_i + 1$. Cette inégalité rend fausse soit la sixième condition du prédicat $\text{Onelocal fault}(k, d, k')$, soit sa troisième condition si le voisinage de p_k est réduit à p_i . Par conséquent, la règle \mathbf{O}_k n'est pas applicable, et la règle

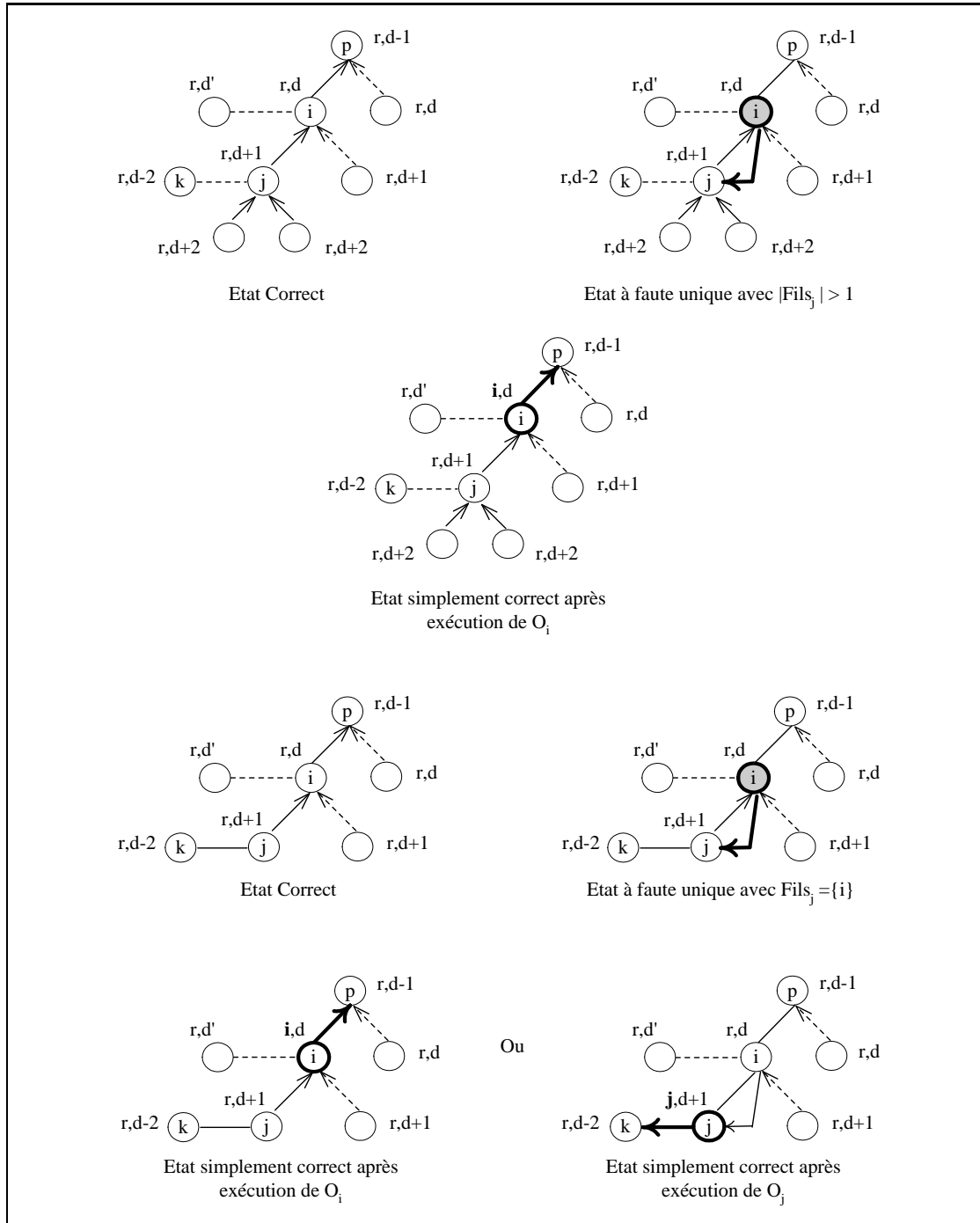


FIG. 5.9 – Un exemple du cas 5a

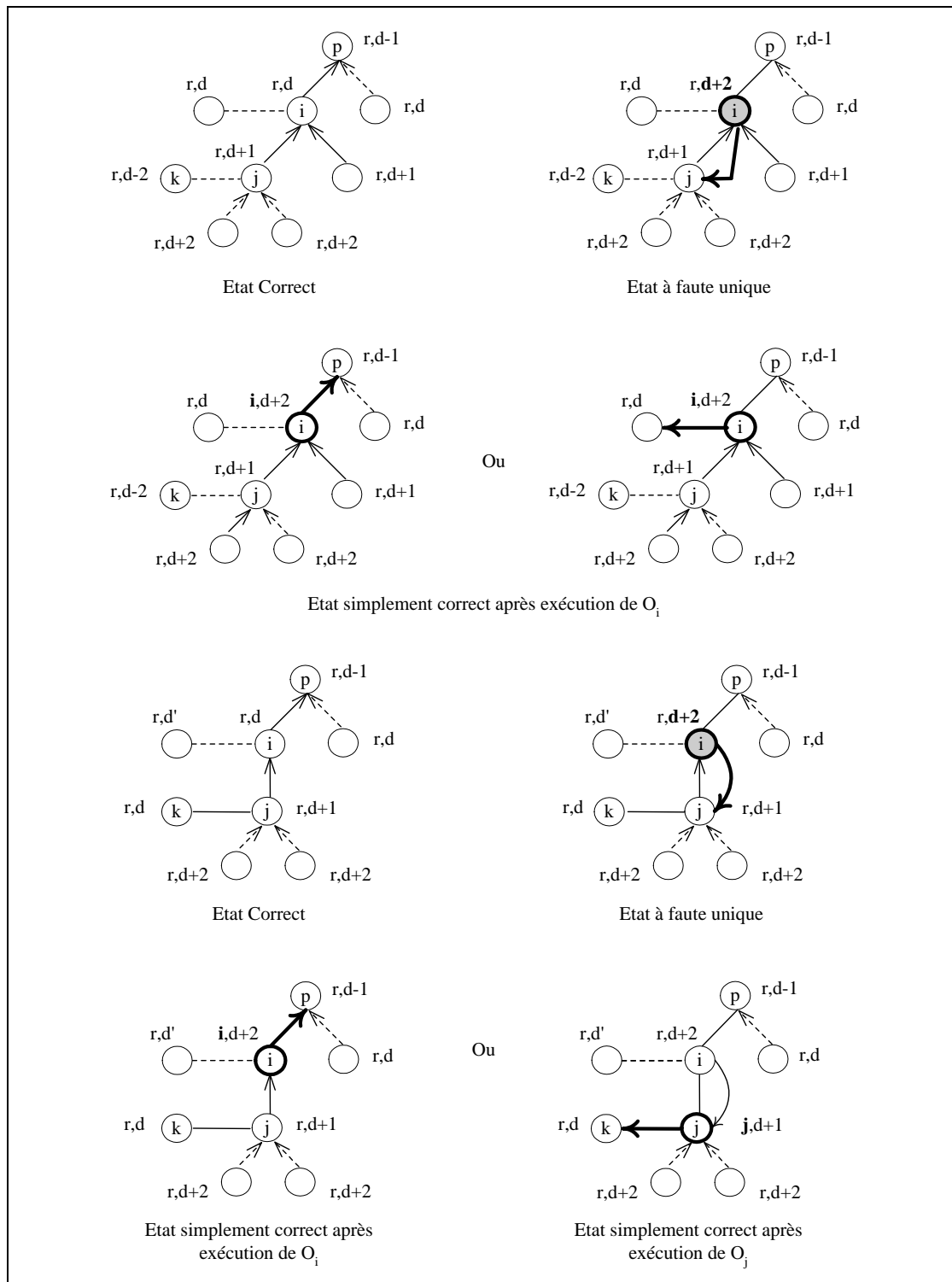


FIG. 5.10 – Un exemple du cas 5b

\mathbf{B}_k non plus car $Tree(k)$ est vérifié. En outre, nous constatons que la sixième et la septième condition de $Onelocalfault(i, d, k)$ sont vérifiées. En effet, pour tout $k \in Fils_i$, $dist_k \neq dist_i + 1$ et les deux assertions de l'implication de la septième condition sont vraies. Nous distinguons deux cas selon le nombre de fils de i :

- i. $|Fils_i| > 1$: d'une part, le prédicat $Onelocalfault(i, d, k)$ est vérifié car l'assertion à droite de l'implication de la cinquième condition est vraie. Par conséquent, la règle \mathbf{O}_i est exécutable mais les règles \mathbf{B}_i et \mathbf{C}_i ne le sont pas. D'autre part, le prédicat $Onelocalfault(j, d, k')$ n'est pas vérifié car la septième condition ne l'est pas. En fait, l'assertion à gauche de ladite implication est vraie mais celle de droite ne l'est pas. Ainsi, nous nous trouvons dans un cas analogue à celui décrit dans 4(a). Par conséquent, il existe un k tel que la règle $\mathbf{O}_i(k)$ soit exécutable et son exécution conduit à un état du lemme 5.6. Un exemple est représenté sur la figure 5.10.
 - ii. $Fils_i = \{j\}$: dans ce cas, les deux prédicats $Onelocalfault(j, d, k')$ et $Onelocalfault(i, d, k)$ sont évalués à vrai. Pour le deuxième prédicat, c'est l'assertion à droite de l'implication de la cinquième condition qui est toujours vraie puisque $par_i = j$. Par conséquent, la règle \mathbf{O}_i est exécutable mais les règles \mathbf{B}_i et \mathbf{C}_i ne le sont pas. De façon analogue au cas 4(c), la règle \mathbf{O}_i est exécutable et \mathbf{O}_j est la seule autre règle éventuellement exécutable. L'exécution de l'une de ces deux règles rendra la deuxième non applicable et conduira le système à un état du lemme 5.6 vis-à-vis de i ou de j . Un exemple est représenté sur la figure 5.10.
- (c) $dist_i \neq dist_j + 1$ et $dist_j \neq dist_i + 1$: notons que dans ce cas, nous considérons que $|dist_j - dist_i| \neq 1$. Pour tout $k \in Fils_i \setminus j$, la distance $dist_k$ est différente de $dist_i + 1$. Cette inégalité rend fausse soit la sixième condition de $Onelocalfault(k, d, k')$, soit sa troisième condition si le voisinage de p_k est réduit à p_i . Par conséquent la règle \mathbf{O}_k n'est pas applicable et \mathbf{B}_k non plus car $Tree(k)$ est vérifié. En outre, nous constatons que la cinquième, la sixième et la septième condition de $Onelocalfault(i, d, k)$ sont vérifiées. En effet, $dist_i \neq dist_j + 1, \forall k \in Fils_i, dist_j \neq dist_i + 1$, et les deux assertions de l'implication de la septième condition sont vraies. Par conséquent, la règle \mathbf{O}_i est exécutable mais les règles \mathbf{B}_i et \mathbf{C}_i ne le sont pas. Nous distinguons trois cas selon le nombre de fils de p_i et de p_j :

- i. $|Fils_i| > 1$: dans ce cas, le prédicat $Onelocalfault(j, d, k')$ n'est pas vérifié car la septième condition ne l'est pas. En fait, l'assertion à gauche de ladite

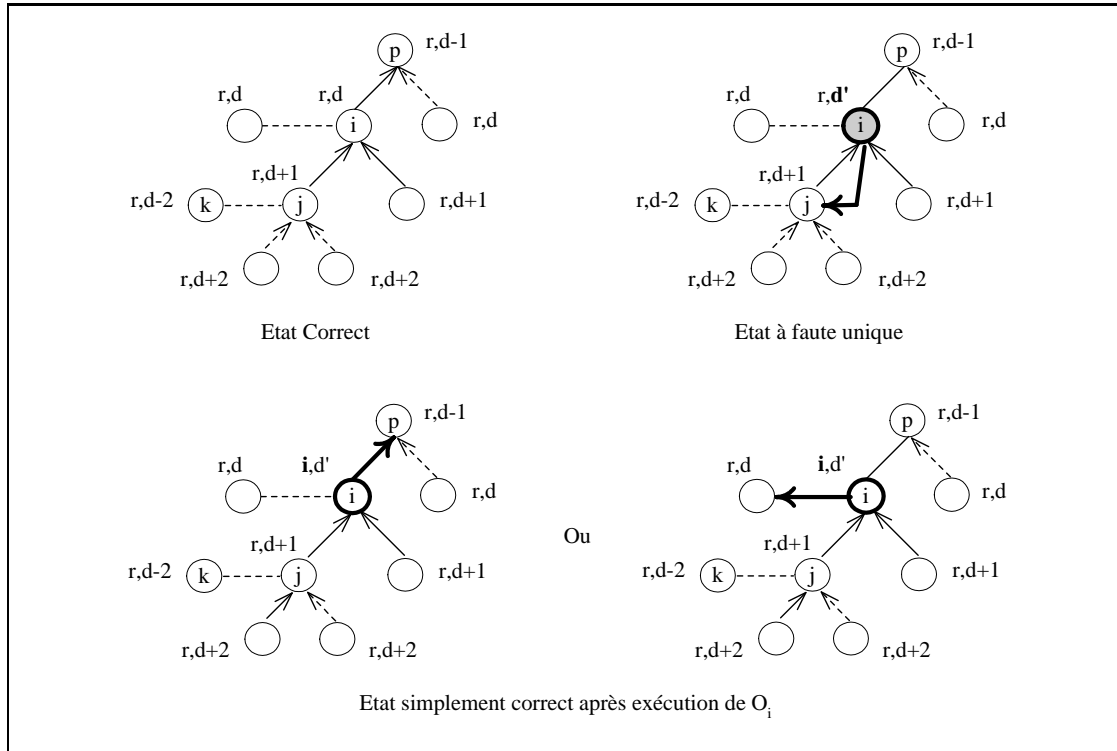


FIG. 5.11 – Un exemple du cas $5ci$

implication est vraie alors que celle de droite ne l'est pas. Ainsi, nous nous trouvons dans un cas analogue à celui décrit dans 4(a). Par conséquent, il existe un k tel que la règle $O_i(k)$ soit exécutable et son exécution conduit à un état du lemme 5.6. Un exemple est représenté sur la figure 5.11.

- ii. $Fils_i = \{j\}$ et $Fils_j = \{i\}$: dans ce cas, le prédicat $Onelocal\ fault(j, d, k')$ est vérifié. De façon analogue au cas décrit dans 4(c), la règle O_i est exécutable et O_j est la seule autre règle éventuellement exécutable. L'exécution de l'une de ces deux règles rendra la deuxième non applicable et conduira le système à un état du lemme 5.6 vis-à-vis de i ou de j . Un exemple est représenté sur la figure 5.12.
- iii. $Fils_i = \{j\}$ et $|Fils_j| > 1$: dans ce cas, le prédicat $Onelocal\ fault(j, d, k')$ n'est pas vérifié car la quatrième condition ne l'est pas. En fait, il existe un d' tel que $\forall k \neq i \in Fils_i, dist_k = d'$ alors que $dist_i \neq d'$. Ainsi, nous nous trouvons dans un cas analogue à celui décrit dans 4(a). Par conséquent, il existe un k tel que la règle $O_i(k)$ soit exécutable et son exécution conduit à un état du lemme 5.6. Un exemple est représenté sur la figure 5.13. ■

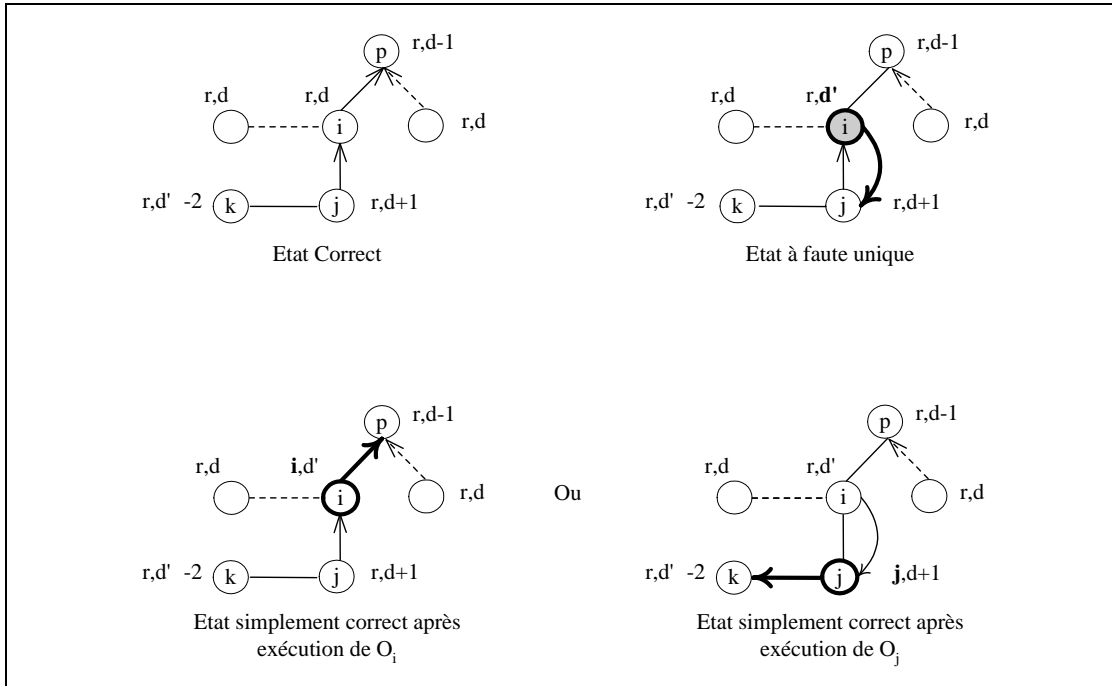


FIG. 5.12 – Un exemple du cas 5cii

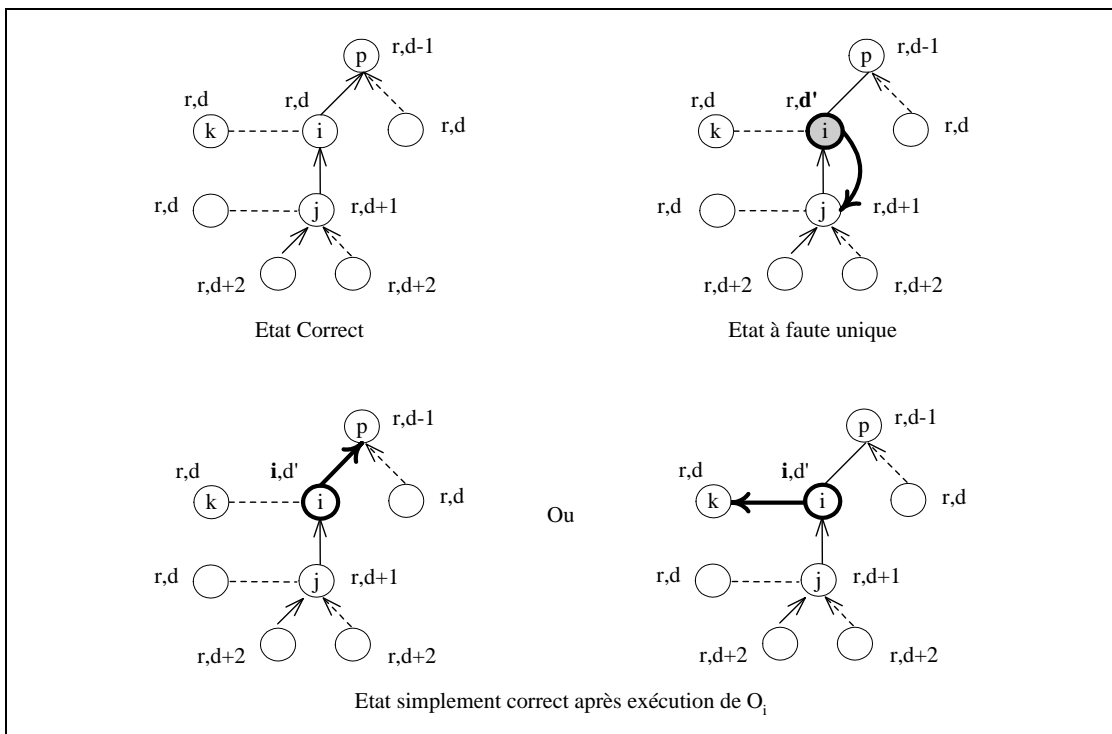


FIG. 5.13 – Un exemple du cas 5ciii

Avant de commencer la preuve, nous définissons quelques notations que nous utilisons dans la suite. Dans l'état s_n , nous dirons que :

- un processeur p_i possède un grant du processeur racine d'identité r pour le processeur p_j si et seulement si $req_i^n = j$ et $root_i^n = r$ et $dir_i^n = grant$;
- un processeur p_i est dit *attaché* à r si et seulement si $root_i^n = r$.

Vis-à-vis d'une action a_n , nous dirons que :

- un processeur p_i se rattache à r , par l'intermédiaire du processeur p_j , si et seulement si $root_i^n \neq r$ et $root_i^{n+1} = r$ avec $par_i^{n+1} = j$;
- un processeur p_i se détache de r si et seulement si $root_i^n = r$ et $root_i^{n+1} \neq r$;
- un processeur p_j transmet un grant de r à p_i si et seulement si a_n est l'action R_i avec j comme paramètre et $root_i = r$;
- un processeur p_i maintient une requête pour r vers p_j si et seulement si $Asks(i, j)$ est vérifié et $root_j = r$.

5.6.1 Disparition des fausses identités

Dans cette sous-section, nous montrons que toute identité, contenue dans l'une des variables d'un processeur ne correspondant pas à l'identité d'un des processeurs du réseau, finira par disparaître.

Lemme 5.9 *Soit r une identité absente du réseau et p_i un processeur, alors si aucun grant de p_r pour p_i n'est présent dans un état d'une exécution, un tel grant ne sera jamais créé dans le futur.*

Preuve. Examinons les règles qui conduisent à la création sur un processeur p_j d'un grant de p_r pour p_i . Ceci ne peut provenir que d'un changement d'une des variables $root_j$, req_j ou dir_j .

- l'exécution de la règle **O** _{j} ne peut créer un tel grant car après son application $dir_j = ask$;
- l'exécution de la règle **B** _{j} ne peut créer un tel grant car après son application $dir_j = \perp$;
- l'exécution de la règle **A** _{j} ne peut créer un tel grant car après son application $dir_j = ask$;
- l'exécution de la règle **J** _{j} ne peut créer un tel grant car après son application $dir_j = \perp$;
- l'exécution de la règle **C** _{j} ne peut créer un tel grant car après son application $dir_j = \perp$;
- l'exécution de la règle **F** _{j} ne peut créer un tel grant car après son application $dir_j = ask$;

- l'exécution de la règle \mathbf{G}_j ne peut créer un tel grant car pour qu'elle puisse être appliquée, il faut que $root_j = j \neq r$ et $root_j$ reste inchangée suite à l'exécution de \mathbf{G}_j ;
- l'exécution de la règle \mathbf{R}_j ne peut créer un tel grant car pour qu'elle puisse être appliquée il faut que $root_j = root_{par_j}$, $req_j = req_{par_j}$ et $dir_{par_j} = grant$. Or, comme un tel grant n'existe pas avant l'exécution de la règle, nous avons soit $root_j \neq r$, soit $req_j \neq i$. De plus, ces deux dernières variables ne sont pas modifiées par l'exécution de \mathbf{R}_j .

■

Lemme 5.10 *Soit r une identité absente du réseau et p_i un processeur. Si un état survient dans lequel p_i est attaché à r , alors p_i se détachera de r dans un état futur de l'exécution.*

Preuve. Supposons qu'il n'en soit pas ainsi et qu'à partir d'un certain état s_n , p_i reste attaché indéfiniment à r . Nous remarquons d'abord que $dist_i$ restera inchangé dans le futur car les seules actions qui modifient la distance sont :

- l'action de la règle \mathbf{B}_i qui positionne $root_i$ à i et donc détacherait p_i de r ;
- l'action de la règle \mathbf{J}_i qui positionne $root_i$ à $root_{ownrmax(i)} \neq root_i$ et donc détacherait p_i de r .

Posons pour $m \geq n$, $dst^m = (-dist^m, nb^m)$ où dans l'état s_m , $dist^m$ représente la distance minimale d'un processeur attaché à r , et nb^m le nombre de processeurs de distance minimale attachés à r . Dans la suite, dst est la suite des valeurs dst^m . Muni de l'ordre lexicographique, le domaine des valeurs possibles de dst n'admet pas de suite infinie strictement décroissante. En effet, il est borné inférieurement par $(-dist_i^n, 1)$ puisque p_i reste attaché indéfiniment à r . Nous allons exhiber une contradiction en montrant que dst^m décroît infiniment souvent et ne croît jamais.

Pour que dst croisse, il faudrait qu'un processeur se rattache à r avec une distance inférieure ou égale à la distance minimale (courante) à r , ou qu'un processeur reste attaché à r en diminuant sa distance. La deuxième alternative est exclue pour les mêmes raisons que pour le processeur p_i . Intéressons-nous au rattachement à r . Trois actions modifient $root_k$: l'action de la règle \mathbf{O}_k , celle de \mathbf{B}_k qui la positionne à $i \neq r$ et celle de \mathbf{J}_k qui est telle qu'après son application $dist_k = dist_{par_k} + 1$ et $root_k = root_{par_k}$. Donc l'exécution de la règle \mathbf{J}_k ne rattache jamais un site à distance minimale.

Montrons maintenant que dst décroît infiniment souvent. Soit p_{i^*} le processeur attaché à r dans l'état s_n de distance minimale, deux cas se présentent :

- $dist_{i^*} > 0$: dans ce cas, puisque cette distance est minimale, soit $root_{par_{i^*}} \neq r$, soit $dist_{par_{i^*}} \neq dist_{i^*} - 1$;
- $dist_{i^*} = 0$: dans ce cas, $root_{i^*} = r \neq i^*$ et $Asks(i^*, j)$ n'est vérifié pour aucun j .

Dans les deux cas $Tree(i^*)$ n'est pas vérifié. D'autre part, supposons que $par_{i^*} \neq i^*$. Il ne peut exister de voisin p_{j^*} de $p_{par_{i^*}}$ tel que $dist_{j^*} = dist_{i^*} - 2$ et $root_{j^*} = r$ en raison de la définition de p_{i^*} . De plus, $par_{i^*} \neq r$ et donc le prédicat $Oneparfault(i^*)$ n'est pas vérifié. En conséquence, par examen des préconditions, soit \mathbf{O}_{i^*} , soit \mathbf{B}_{i^*} est applicable et aucune autre règle n'est applicable à p_{i^*} . De plus, ceci reste vrai tant qu'aucune de ces deux règles n'est appliquée. Puisque l'exécution est équitable, $\exists n_0 \geq n$ tel que a_{n_0} soit l'une de ces actions et tel que $root_{i^*} = i^*$ dans s_{n_0+1} . Ceci induit que p_{i^*} se détache de r . Donc dst décroît.

En itérant ce raisonnement à partir de n_0 , on en déduit que dst décroît infiniment souvent ce qui est contradictoire. ■

Lemme 5.11 *Soit r une identité absente du réseau et p_i un processeur, alors si aucun grant de r pour p_i n'est présent dans un état d'une exécution, p_i ne se rattachera jamais à r dans le futur.*

Preuve. Les trois actions qui modifient la valeur de la racine de p_i sont celles des règles \mathbf{O}_i , \mathbf{B}_i et \mathbf{J}_i . Les deux premières actions positionnent la racine à une identité du réseau. La troisième action nécessite que par_i possède un grant de r pour p_i . ■

Proposition 5.12 *Soit r une identité absente du réseau. Dans toute exécution, il existe un état à partir duquel aucun processeur ne sera jamais plus rattaché à r .*

Preuve. Supposons que ce ne soit pas le cas et appelons J le sous-ensemble des processeurs qui sont attachés infiniment souvent à r . Posons s_n un état à partir duquel seuls les processeurs de J seront attachés à r . D'après le lemme 5.10, ces processeurs se détacheront et se rattacheront infiniment souvent à r . D'après les lemmes 5.9 et 5.11, il doit exister un *grant* de r pour chacun de ces processeurs dans tout état à partir de s_n . Mais ces *grants* sont nécessairement possédés par des processeurs de J . Autrement dit, dans tout état à partir de s_n , chaque processeur de J possède un grant de r pour un autre processeur de J et ceci constitue une bijection de J dans J . Posons s_m le premier état après s_n pour lequel a_m est le rattachement de l'un des processeurs de J à r . Ceci ne peut se faire que par l'exécution de la règle \mathbf{J}_i , ce qui signifie que p_i ne possédait pas de *grant* pour un autre site dans s_m puisque dir_i devait être positionné à *ask*. Ceci est une contradiction. ■

Nous avons donc démontré que toute exécution contenait un suffixe dans lequel aucun processeur n'est jamais attaché à une identité absente du réseau. Dans la suite, nous appellerons *1-suffixe* de l'exécution, un tel suffixe.

5.6.2 Maintien d'une requête ou couverture de l'arbre

Dans cette sous-section, nous montrons que tout processeur ne faisant pas partie de l'arbre stabilisé maintiendra une requête vers un voisin appartenant à l'arbre.

Définition 5.3 Soit s_n un état d'une exécution et p_r le processeur de plus grande identité, alors $Arbre_n$ est l'arbre éventuellement vide dont l'ensemble des processeurs est défini récursivement par :

- $p_r \in Arbre_n$ si $root_r^n = r$, $par_r^n = r$ et $dist_r^n = 0$
- $p_i \neq p_r \in Arbre_n$ si $root_i^n = r$, $par_i^n \neq i \in Arbre_n$ et $dist_i^n = dist_{par_i}^n + 1$

La structure de l'arbre est déduite par l'ensemble des variables par_i .

Nous appelons *Arbre*, la suite des $Arbre_n$. Nous dirons qu'un processeur se rattache à *Arbre* s'il n'appartenait pas à *Arbre* dans l'état précédent et qu'il appartient à *Arbre* dans l'état courant. De plus, nous dirons qu'un processeur se détache de *Arbre* s'il appartenait à *Arbre* dans l'état précédent et qu'il n'appartient plus à *Arbre* dans l'état courant.

Lemme 5.13 Dans tout 1-suffixe d'une exécution, soit p_r reste attaché en permanence à *Arbre*, soit p_r se rattache une fois à *Arbre* et ne se détachera plus.

Preuve : Supposons que p_r ne soit pas attaché à *Arbre* dans un état d'un 1-suffixe de l'exécution. Ceci signifie que $root_r \neq r$ ou $par_r \neq r$ ou $dist_r \neq 0$. Dans tous les cas $Tree(r)$ et $Asks(r, k)$ ne sont pas vérifiés. De plus, $\forall(d, j)$ le prédicat $Onelocalfault(r, d, j)$ n'est pas vérifié. Pour voir cela, il suffit de remarquer que r est la plus grande identité présente dans les variables. Pour les mêmes raisons, si $par_i \neq i$ alors le prédicat $Oneparfault(i)$ n'est pas vérifié. En conséquence, la règle \mathbf{B}_r est la seule applicable. En vertu de l'équité, la règle \mathbf{B}_r sera exécutée et p_r se rattachera à *Arbre*. De plus, une fois ce rattachement effectué, aucune des règles \mathbf{O}_r , \mathbf{B}_r et \mathbf{J}_r n'est plus jamais applicables à p_r , et ce dernier ne se détachera jamais de *Arbre*. ■

Lemme 5.14 Dans tout 1-suffixe d'une exécution, tout noeud qui se rattache à *Arbre* ne se détachera pas de *Arbre* avant que son père ne se détache (éventuellement) de *Arbre*.

Preuve. La seule action qui permet à un processeur p_i de se rattacher à *Arbre* est celle de la règle \mathbf{J}_i . Or, elle implique que $\forall k \in Filis_i, root_k = r \Rightarrow dist_k = dist_i + 1$. Supposons que le processeur père de p_i d'identité par_i ne se détache pas de *Arbre*, dans ce cas sa distance ne se modifie pas :

- si $par_i \neq r$, une modification de la distance serait due à O_{par_i} ou B_{par_i} qui détacherait par_i de *Arbre*;
- si $par_i = r$ alors c'est une conséquence de la définition de *Arbre*.

En conséquence, la seule action qui pourrait détacher p_i de *Arbre* est l'exécution de la règle \mathbf{O}_i . Cependant un fils p_k de p_i qui a r comme valeur de la racine est à bonne distance au moment du rattachement. Il ne peut changer sa distance qu'en modifiant $root_k$, rendant ainsi la règle \mathbf{O}_i non applicable. Un processeur qui adopterait p_i comme père avec $root_k = r$ ne peut le faire qu'en exécutant \mathbf{J}_k et se trouve aussi à la bonne distance. En résumé, si le processeur p_{par_i} ne se détache pas de l'arbre alors \mathbf{O}_i n'est jamais applicable et p_i restera attaché à *Arbre*. ■

Lemme 5.15 *Dans toute exécution, $\exists n$ tel que $\forall m, m' \geq n$ $Arbre_m = Arbre_{m'}$.*

Preuve. D'après le lemme 5.13, à partir d'un certain état le processeur racine d'identité r appartient toujours à *Arbre*. Dans le suffixe qui suit cet état, examinons les processeurs de profondeur 1 de *Arbre*. Ceux-ci peuvent se détacher, mais tout processeur qui se rattachera à une profondeur 1 ne se détachera plus en vertu du lemme 5.14. Autrement dit, à partir d'un certain état, le sous-arbre de profondeur 1 est stabilisé. Par récurrence, et sachant que la profondeur maximale dans l'arbre est strictement majorée par le nombre de processeurs, on en conclut que l'arbre se stabilisera. ■

Par conséquent, $Arbre_n$ est une structure d'arbre qui se stabilise dans toute exécution. Par abus de langage, nous appellerons *Arbre* l'arbre stabilisé d'une exécution. De plus, nous appellerons $Arbre^c$ l'ensemble des processeurs n'appartenant pas à *Arbre*.

Lemme 5.16 *Dans toute exécution, après stabilisation de *Arbre*, il existe un état à partir duquel $\forall p_i \in Arbre^c$, soit $p_{par_i} \in Arbre^c$ soit $root_i \neq r$.*

Preuve. Supposons que se ne soit pas le cas et qu'après stabilisation de *Arbre*, il existe un état s_n dans lequel $\forall p_i \in Arbre^c$, $p_{par_i} \in Arbre$ et $root_i^n = r$.

D'après la définition de *Arbre*, $dist_i^n \neq dist_{par_i}^n + 1$. En outre, le processeur p_{par_i} ne se détache plus de r puisqu'il est attaché à l'arbre stabilisé. Une modification de la distance serait due à l'exécution de la règle \mathbf{O}_{par_i} qui détacherait le processeur p_{par_i} de *Arbre*. En conséquence, \mathbf{O}_i et \mathbf{B}_i sont les seules règles exécutables. Si la règle \mathbf{B}_i est exécutée alors $\exists m \geq n$ tel que $root_i^m = i \neq r$ et $p_{par_i}^m = i \in Arbre^c$. Dans le cas contraire, l'exécution de \mathbf{O}_i détachera p_i de l'arbre $root_i^m = i \neq r$.

Montrons que cette propriété reste toujours vérifiée. Autrement dit, montrons que $\forall m' \geq m$ tel que dans tout état $s_{m'}$, $\forall p_i \in Arbre^c$ soit $p_{par_i}^{m'} \in Arbre^c$, soit $root_i^{m'} \neq r$:

- si $root_i^m \neq r$: dans ce cas $root_i$ ne peut être égale à r que si p_i se rattache à un père de racine r .
- si $root_i^m = r$: dans ce cas, $p_{par_i^m} (\in Arbre^c)$ s'est attaché à l'arbre en exécutant la règle \mathbf{J}_{par_i} qui oblige tout fils p_k ayant r comme valeur de la racine à être à bonne distance. Étant l'un de ces fils et ayant pour racine r , le processeur p_i est à la bonne distance de son père, ce qui est contradictoire. ■

Dans la suite, nous appellerons *2-suffixe* d'une exécution, le suffixe du 1-suffixe dans lequel *Arbre* est stabilisé et le lemme 5.16 est vérifié.

Lemme 5.17 *Dans un 2-suffixe,*

1. aucun processeur ne se rattachera à r par un processeur de *Arbre* ;
2. aucun grant de r ne sera transmis par un processeur de *Arbre* à un processeur de *Arbre^c* ;
3. soit p_i un processeur de *Arbre^c*, si dans un état aucun processeur de *Arbre^c* ne possède de grant de r pour p_i , alors aucun processeur de *Arbre^c* ne possédera dans le futur un grant de r pour p_i .

Preuve. La première affirmation est une conséquence de la stabilité de *Arbre*. La deuxième affirmation provient du fait qu'un *grant* est transmis à un fils pour lequel *Tree* est vérifié, ce qui n'est pas possible ici car sinon ce processeur appartiendrait à *Arbre*. De manière similaire à la preuve du lemme 5.9, la troisième affirmation découle de l'examen des règles en s'appuyant sur la deuxième affirmation et sur le fait que le processeur $p_r \in Arbre$. ■

Lemme 5.18 *Soit r la plus grande identité du réseau. Supposons qu'il existe $p_i \in Arbre^c$, $p_j \in Arbre$ et $\exists n$ tel que dans tout 2-suffixe de l'exécution, pour l'état s_n , si p_i maintient une requête pour r vers p_j alors pour tout état s_m avec $m \geq n$, p_i maintient une requête pour r vers p_j .*

Preuve. Il s'agit de montrer que *Asks*(i, j) ne peut pas devenir faux. Tout d'abord, p_i ne peut pas avoir de nouveau fils p_k . En effet, seule l'exécution des règles $\mathbf{O}_k(i)$ ou \mathbf{J}_k pourrait conduire à cette situation.

Les fils de p_i ne peuvent changer de distance sans changer de père. Autrement dit, la condition sur les fils incluse dans *Asks*(i, j) restera vérifiée. De même, cette condition reste vérifiée, pour *Sat*(j), puisqu'on est dans un 2-suffixe et que $j \in Arbre$. ■

Lemme 5.19 *Soit r la plus grande identité du réseau. Dans tout 2-suffixe de l'exécution :*

1. soit $Arbre$ couvre l'ensemble des processeurs ;
2. soit $\exists p_i \in Arbre^c$, $\exists p_j \in Arbre$ et $\exists n$ tel que dans un état s_m avec $m \geq n$, p_i maintient une requête pour r vers p_j ;
3. soit $\forall p_i \in Arbre^c$, si dans un état p_i est attaché à r , alors p_i se détachera de r dans un état futur de l'exécution.

Preuve. Supposons qu'aucune des trois propriétés ne soit vérifiée et en particulier qu'à partir d'un certain état s_n du 2-suffixe de l'exécution, $p_i \in Arbre^c$ reste attaché indéfiniment à r .

Nous remarquons d'abord que $dist_i$ restera inchangée dans le futur car les seules actions qui modifient la distance sont :

- l'exécution de la règle \mathbf{B}_i qui positionne $root_i$ à i mais qui détacherait p_i de r
- l'exécution de la règle \mathbf{J}_i qui n'est pas applicable puisque $ownrmax(i) = i$

Posons pour $m \geq n$, $dst^m = (-dist^m, nb^m)$ où dans l'état s_m , $dist^m$ est la distance minimale d'un processeur appartenant à $Arbre^c$ attaché à r et nb^m est le nombre de processeurs de $Arbre^c$ attachés à r de distance minimale. Dans la suite, dst est la suite des valeurs dst^m . Muni de l'ordre lexicographique, le domaine des valeurs possibles de dst n'admet pas une suite infinie strictement décroissante. En effet, il est borné inférieurement par $(-dist_i^n, 1)$ puisque p_i reste attaché indéfiniment à r . Nous allons exhiber une contradiction en montrant que dst^m décroît infiniment souvent et ne croît jamais.

Pour que dst croisse, il faudrait qu'un processeur se rattache à r avec une distance inférieure ou égale à la distance minimale à r courante, ou qu'un processeur reste attaché à r en diminuant sa distance. La deuxième alternative est exclue pour les mêmes raisons que pour le processeur p_i . Intéressons-nous au rattachement à r . Trois actions modifient $root_k$: l'action de la règle \mathbf{O}_k , celle de \mathbf{B}_k qui la positionnent à $i \neq r$ et celle de \mathbf{J}_k qui est telle qu'après son application $dist_k = dist_{par_k} + 1$ et $root_k = root_{par_k}$. Or le processeur correspondant au paramètre j dans J_k ne peut appartenir à $Arbre$ car sinon le site se rattacherait à $Arbre$. Donc \mathbf{J}_k ne rattache jamais un site à une distance minimale.

Montrons maintenant que dst décroît infiniment souvent. Soit $p_{i^*} \in Arbre^c$ le processeur attaché à r dans l'état s_n de distance minimale, deux cas se présentent :

- $dist_{i^*} > 0$: dans ce cas, puisque cette distance est minimale, soit $root_{par_{i^*}} \neq r$, soit $dist_{par_{i^*}} \neq dist_{i^*} - 1$;
- $dist_{i^*} = 0$: dans ce cas, $root_{i^*} = r \neq i^*$ et $Asks(i^*, j)$ n'est vérifié pour aucun j .

Dans les deux cas $Tree(i^*)$ n'est pas vérifié. D'autre part, supposons que $par_{i^*} \neq i^*$. Un processeur p_{j^*} voisin de $p_{par_{i^*}}$ qui serait tel que $dist_{j^*} = dist_{i^*} - 2$ et $root_{j^*} = r$ appartient nécessairement à $Arbre$ en raison de la définition de p_{i^*} . De plus, comme $par_{i^*} \neq r$ et en vertu du lemme 5.1, soit le prédicat $Oneparfault(i^*)$ n'est pas vérifié, soit il existe

un couple (d, j) pour lequel le prédicat $Onelocal\ fault(par_{i^*}, d, j)$ est vérifié. Supposons que les règles $\mathbf{O}_{i^*}(j)$ (pour un p_j quelconque) et \mathbf{B}_{i^*} ne soient jamais exécutées et que $Onepar\ fault(i^*)$ soit constamment vérifié à partir d'un certain état. Par conséquent il existe un couple (d, j) pour lequel $Onelocal\ fault(par_{i^*}, d, j)$ sera constamment vérifié. Donc pour un certain p_j , le prédicat $Asks(par_{i^*}, j)$ sera constamment vérifié dans le futur (voir le lemme 5.18). Cela qui signifie que le processeur $p_{par_{i^*}}$ maintient une requête pour un processeur de *Arbre*. Ceci est contradictoire avec nos hypothèses du début de la preuve. Donc, infiniment souvent $Onepar\ fault(i^*)$ ne sera pas vérifiée. En conséquence, infiniment souvent soit la règle \mathbf{O}_{i^*} soit la règle \mathbf{B}_{i^*} sera applicable (aucune autre règle n'étant applicable à p_{i^*}). Par équité, l'une de ces deux règles sera appliquée et $\exists n_0 \geq n$ tel que a_{n_0} soit l'une de ces actions et tel que $root_{i^*} = i^*$ dans s_{n_0+1} . Ceci induit que p_{i^*} se détachera de r et par conséquent dst décroît.

En itérant ce raisonnement à partir de n_0 , on en déduit que dst décroît infiniment souvent ce qui est contradictoire. ■

Lemme 5.20 *Soit r la plus grande identité du réseau. Dans tout 2-suffixe de l'exécution :*

1. *soit $Arbre$ couvre l'ensemble des processeurs ;*
2. *soit $\exists p_i \in Arbre^c, \exists p_j \in Arbre$ et $\exists n$ tel que dans un état s_m avec $m \geq n$, p_i maintient un requête pour r vers p_j ;*
3. *soit $\exists n$ tel que dans un état s_m avec $m \geq n, \forall p_i \in Arbre^c, p_i$ n'est pas attaché à r .*

Preuve. Supposons qu'aucune des trois hypothèses ne soit vérifiée. Appelons J le sous-ensemble des processeurs n'appartenant pas à *Arbre* qui sont attachés infiniment souvent à r . Posons s_n un état à partir duquel seuls les processeurs de J seront attachés à r parmi ceux n'appartenant pas à *Arbre*. D'après le lemme 5.19, ces processeurs se détacheront et se rattacheront infiniment souvent à r . D'après le lemme 5.17, il doit exister un grant de r pour chacun de ces processeurs dans tout état à partir de s_n possédé par un processeur de J . Autrement dit, dans tout état à partir de s_n , chaque processeur de J possède un grant de r pour un autre processeur de J et ceci constitue un bijection de J dans J . Posons s_m le premier état après s_n pour lequel a_m est le rattachement de l'un des processeurs de J à r . Ceci ne peut se faire que par J_i , ce qui signifie que p_i ne possédait pas de *grant* pour un autre processeur dans s_m puisque dir_i devait être positionné à *ask*. Ceci est une contradiction. ■

Proposition 5.21 *Soit r la plus grande identité du réseau. Dans tout 2-suffixe de l'exécution :*

1. *soit $Arbre$ couvre l'ensemble des processeurs ;*

2. soit $\exists p_i \in \text{Arbre}^c$, $\exists p_j \in \text{Arbre}$ et $\exists n$ tels que dans tout état s_m avec $m \geq n$, p_i maintient une requête pour r vers p_j .

Preuve. Supposons que la troisième propriété du lemme précédent soit vérifiée : il existe un rang n tel que dans un état s_m avec $m \geq n$, $\forall p_i \in \text{Arbre}^c$, p_i n'est pas attaché à r . Soit un processeur $p_i \in \text{Arbre}^c$ qui possède un voisin $p_{i'} \in \text{Arbre}$ (cette existence est assurée car le graphe est connexe). Si $i' \neq r$, puisque $\text{root}_i < \text{root}_{i'} = \text{root}_{\text{par}_{i'}}$, alors le prédicat $\text{Equalroot}(i')$ n'est pas vérifié. De même si $i' = r$, puisque $\text{root}_i < r$, alors le prédicat $\text{Equalroot}(i')$ n'est pas vérifié. Si $\text{Tree}(i)$ est vérifié, alors la règle \mathbf{A}_i est applicable avec le processeur paramètre de la règle appartenant à Arbre . Supposons maintenant que $\text{Tree}(i)$ ne soit pas vérifié, alors le prédicat $\text{Oneparfault}(i)$ n'est pas vérifié pour les mêmes raisons.

Par ailleurs, deux cas se présentent pour le prédicat $\text{Onelocalfault}(i, d, j)$:

- s'il existe un couple (d, j) pour lequel le prédicat $\text{Onelocalfault}(i, d, j)$ est vérifié, alors nécessairement p_j et tous les voisins de p_i appartiennent à Arbre . Ainsi, la règle \mathbf{O}_i est applicable et elle le reste jusqu'à ce que p_i maintienne une requête pour r vers un processeur $p_j \in \text{Arbre}$;
- si pour tous les couples (d, j) le prédicat $\text{Onelocalfault}(i, d, j)$ est faux, alors c'est la règle \mathbf{B}_i qui est applicable s'il existe $p_k \in \text{Arbre}$ pour lequel le prédicat $\text{Asks}(i, k)$ est vérifié. L'exécution de cette règle rendra $\text{Tree}(i)$ vrai et la règle \mathbf{A}_i applicable. Cette dernière restera constamment applicable jusqu'à ce qu'elle soit exécutée. Ceci conduit aussi à ce que p_i maintienne une requête pour r vers un certain $p_j \in \text{Arbre}$.

Dans les deux cas la requête de p_i n'est jamais satisfaite, puisque cela contredirait la définition de Arbre . ■

5.6.3 Couverture de Arbre

Nous venons de montrer que dans un 2-suffixe de l'exécution, la structure d'arbre du voisinage de tout processeur $p_i \in \text{Arbre}$ est cohérente ($\text{Sat}(i)$ est vérifié). Maintenant, nous nous plaçons dans un 3-suffixe d'une exécution qui est le suffixe d'un 2-suffixe dans lequel la propriété de la proposition 5.21 est vérifiée.

Définition 5.4 Nous appelons un *dem-chemin de requêtes dans Arbre* , une suite maximale de processeurs p_1, \dots, p_K avec $K \geq 1$ tel que :

- $\forall 1 \leq k \leq K$, $\text{req}_k = \text{dem}$ et $\text{dir}_k = \text{ask}$;
- $\forall 1 \leq k \leq K$, $\text{to}_k = \text{par}_k$;
- $\forall 1 \leq k < K$, $\text{par}_k = k + 1$;
- $\forall 1 < k \leq K$, $\text{from}_k = k - 1$ et $\text{from}_1 = \text{dem}$.

De plus, un *dem-chemin de requêtes* est dit *fructueux* si et seulement si le prédicat $\text{Asks}(\text{dem}, p_1)$ est vrai. Sinon, il est dit *infructueux*. Par ailleurs, un *dem-chemin de*

requêtes est dit *acquitté* s'il existe un suffixe non vide contenant un *grant*. Si ce suffixe est vide, alors le *dem*-chemin est dit non-acquitté.

Lemme 5.22 *Soit r la plus grande identité du réseau. Dans tout 3-suffixe de l'exécution :*

1. *soit Arbre couvre l'ensemble des processeurs ;*
2. *soit $\exists n$ tel que si un dem-chemin infructueux existe dans un état alors il ne peut plus exister dans un état futur de l'exécution.*

Preuve. Supposons qu'il n'en soit pas ainsi et qu'à partir d'un certain état s_n , il existe p_1, \dots, p_K pour $K \neq r$, une suite de processeurs de *Arbre* formant un *dem*-chemin infructueux qui reste indéfiniment.

Posons pour $m \geq n$, $dst^m = (nb^m, prf^m)$ où dans l'état s_m , nb^m est le nombre de chemin infructueux existant et prf^m est la profondeur d'un tel chemin (i.e., le nombre de processeurs participants à ce chemin). Dans la suite, dst est la suite des valeurs dst^m . Muni de l'ordre lexicographique, le domaine des valeurs possibles de dst n'admet pas une suite infinie strictement décroissante. En effet, il est borné inférieurement par $(1, k)$ puisque i_1, \dots, i_K reste indéfiniment. Nous allons exhiber une contradiction en montrant que dst^m décroît infiniment souvent et ne croît jamais.

Pour que dst croisse, il faudrait qu'un nouveau *dem*-chemin infructueux se crée ou qu'un processeur vienne s'ajouter à un *dem*-chemin existant, augmentant ainsi sa profondeur. La première alternative est exclue puisqu'un *dem*-chemin infructueux ne peut être créé. Intéressons-nous au rattachement d'un nouveau processeur à un tel *dem*-chemin. Remarquons tout d'abord que $\forall p_k \in \text{Arbre}$, $root_k = r \neq k$, $ownrmax(k) = k$ et que $Sat(k)$ est vérifié. Par conséquent, les règles \mathbf{B}_k , \mathbf{A}_k , \mathbf{J}_k , et \mathbf{G}_k ne sont applicables.

Examinons les règles du processeur p_{K+1} . Étant un processeur de *Arbre*, $Coherent(K+1)$ est vrai et par conséquent $Onelocalfault(K+1, d, j)$ n'est pas vérifié. Donc la règle \mathbf{O}_k n'est pas applicable. Ainsi les seules règles éventuellement applicables sont : \mathbf{C}_{K+1} , \mathbf{F}_{K+1} et \mathbf{R}_{K+1} . Deux cas se présentent :

- soit $Idle(K+1)$ n'est pas vérifié : dans ce cas \mathbf{F}_{K+1} n'est pas applicable. L'exécution de l'une des deux autres règles n'attachera pas p_{K+1} au *dem*-chemin.
- soit $Idle(K+1)$ est vérifié : dans ce cas $Rgcorrect(K+1)$ et $\neg ForwardsNa(p, i_k)$ sont vérifiés. Par conséquent, les deux règles \mathbf{C}_{K+1} et \mathbf{R}_{K+1} ne sont pas applicables et seule la règle \mathbf{F}_{K+1} est applicable puisque $ForwardsNa(K, K+1)$ est vérifié.

Maintenant, nous étudions les actions possibles des processeurs du *dem*-chemin infructueux. Tous les processeurs p_k pour k allant de 2 à K évaluent leur prédicat $Coherent(k)$ à vrai et par conséquent $Onelocalfault(k, d, j)$ n'est pas vérifié. Donc la règle \mathbf{O}_k n'est pas applicable. De même pour le processeur p_1 puisque son voisin *dem* appartient à *Arbre*^c avec $root_{dem} \neq r$. D'après la définition d'un *dem*-chemin, pour tout processeur p_k le

prédicat $Idle(i_k)$ n'est pas vérifié et par conséquent \mathbf{F}_k n'est pas applicable. Donc, pour tout processeur p_k de ce *dem*-chemin, les seules règles éventuellement applicables sont \mathbf{C}_k et \mathbf{R}_k . Remarquons tout d'abord que du fait que le chemin est infructueux, le prédicat $Asks(dem, 1)$ n'est pas vérifié. Donc $ForwardsNa(dem, i_1)$ est faux et par conséquent $Rgcorrect(1)$ est faux et la règle \mathbf{C}_1 est applicable. Selon les propriétés du *dem*-chemin infructueux, deux cas se présentent :

- le *dem*-chemin est infructueux et non acquitté : dans ce cas, le suffixe contenant un *grant* est vide. Autrement dit, les variables *dir* de tous les processeurs sont valuées par un *ask* et par conséquent la règle \mathbf{R}_k n'est pas applicable. De plus, puisque le prédicat $ForwardsNa(i, j)$ est vrai pour i allant de 2 à K alors le prédicat $Rgcorrect$ est vrai pour tous ces processeurs et par conséquent la règle \mathbf{C}_k n'est pas applicable pour l'ensemble des processeurs $\{p_2, \dots, p_K\}$. En résumé, seule la règle \mathbf{C}_1 est applicable.
- le *dem*-chemin est infructueux et acquitté : dans ce cas, il existe un suffixe contenant un *grant* non vide. Supposons que ce suffixe soit constitué des processeurs p_{i+1}, \dots, p_k . Pour tout processeur p_k de ce suffixe, la valeur de dir_k est à *grant* et par conséquent la règle \mathbf{R}_k n'est pas applicable. De plus, pour tout processeur p_k du suffixe sauf p_{i+1} , le prédicat $Rgcorrect(k)$ est faux et par conséquent la seule règle applicable pour un processeur p_k pour k allant de $i+2$ à K est \mathbf{C}_k . D'autre part, pour tout processeur p_k pour k allant de i à 2, le prédicat $Rgcorrect(k)$ est vrai (puisque $ForwardsNa(i, j)$ est vrai) et par conséquent la règle \mathbf{C}_k n'est pas applicable. De plus, le bout de chemin formé par les processeurs p_2, \dots, p_{i-1} est un *dem*-chemin infructueux et non acquitté. D'après le cas précédent, pour tout processeur p_k sur ce bout de chemin, la règle \mathbf{R}_k n'est pas applicable. Pour le processeur p_i , puisque $dir_{i+1} = grant$ et que $ForwardsNa(i, i+1)$ est vrai, la règle \mathbf{R}_i est applicable.

Montrons maintenant que *dst* décroît infiniment souvent. Soit p_1, \dots, p_K le *dem*-chemin infructueux dans l'état s_n et qui reste indéfiniment. Deux cas se présentent :

- le *dem*-chemin est infructueux et acquitté : dans ce cas, les règles applicables sont \mathbf{C}_1 , \mathbf{R}_i et \mathbf{C}_k pour k allant de $i+2$ à K . L'exécution de ces règles n'attachera aucun processeur au *dem*-chemin mais détachera $K - i - 2$ processeurs. Donc *dst* décroît.
- le *dem*-chemin est infructueux et non acquitté : dans ce cas, les règles applicables sont \mathbf{C}_1 et \mathbf{F}_{K+1} . Suite à l'exécution de ces deux règles, p_{K+1} rejoint le *dem*-chemin alors que p_1 s'en détache. La profondeur du *dem*-chemin est la même. L'itération de l'exécution de ces deux règles sur les différents processeurs de l'arbre ne changera pas la profondeur du *dem*-chemin mais le déplacera jusqu'au processeur racine p_r . Ce dernier exécutera alors \mathbf{G}_r qui transformera le *dem*-chemin infructueux non

acquitté en un *dem*-chemin infructueux acquitté décrit dans le cas précédent.

Puisque l'exécution est équitable, $\exists n_0 \geq n$ tel que la procédure de ré-initialisation des variables de requêtes sur tous les processeurs jusqu'à la racine r finira par faire disparaître le *dem*-chemin infructueux. Donc *dst* décroît.

En itérant ce raisonnement à partir de n_0 , on en déduit que *dst* décroît infiniment souvent, ce qui est contradictoire. ■

Corollaire 5.23 *Soit r la plus grande identité du réseau. Dans tout 3-suffixe de l'exécution :*

1. *soit Arbre couvre l'ensemble des processeurs ;*
2. *soit $\exists n$ tel que dans cet état tous les dem-chemins sont fructueux.*

Preuve. La preuve découle du lemme précédent. En effet, un chemin infructueux disparaîtra et aucun ne sera créé dans le futur. Ainsi, à partir d'un certain état s_n du 3-suffixe, tout les *dem*-chemin existant sont fructueux acquittés ou non acquittés. ■

Lemme 5.24 *Soit r la plus grande identité du réseau. Dans tout 3-suffixe de l'exécution :*

1. *soit Arbre couvre l'ensemble des processeurs ;*
2. *soit il existe p_1, \dots, p_K un dem-chemin fructueux non acquitté et alors p_1, \dots, p_K deviendra un dem-chemin fructueux acquitté dans un état futur de l'exécution.*

Preuve. Soit p_1, \dots, p_K un *dem*-chemin fructueux non acquitté. D'après le lemme 5.22, la seule règle applicable est \mathbf{F}_{K+1} . Son exécution rattachera le processeur p_{K+1} au *dem*-chemin. A l'état suivant s_{n+2} , la seule règle applicable est \mathbf{F}_{K+2} et son exécution rattachera le processeur p_{K+2} au *dem*-chemin. Ainsi de suite jusqu'à la racine qui se rattachera au *dem*-chemin en exécutant \mathbf{F}_r . Dans cet état, seule la règle \mathbf{G}_r est applicable et son exécution transformera le *dem*-chemin fructueux non-acquitté en un *dem*-chemin fructueux acquitté. ■

Lemme 5.25 *Soit r la plus grande identité du réseau. Dans tout 3-suffixe de l'exécution :*

1. *soit Arbre couvre l'ensemble des processeurs ;*
2. *soit $\exists p_0 \in \text{Arbre}^c$, $\exists p_1 \in \text{Abre}$ et $\exists n$ tel que dans tout état s_m avec $m \neq n$, p_i rejoint Arbre avec $\text{par}_i = j$*

Preuve. D'après le lemme 5.24, à partir d'un certain état tous les *dem*-chemins fructueux non acquittés deviendront fructueux acquittés. A partir de cet état, sur un le p_0 -chemin (constitué des processeurs p_1, \dots, p_r) et d'après le lemme 5.22, les deux règles applicables

sont \mathbf{C}_r et \mathbf{R}_k (pour $par_k = r$). L'exécution de la première règle met à \perp les variables de requête de la racine et permet à son fils p_k de transmettre le grant à son propre fils via la structure de l'arbre. Dans l'état suivant, \mathbf{C}_k (pour $par_k = r$) et \mathbf{R}_{k-1} (pour $par_{k-1} = k$) sont applicables. L'exécution de ces deux règles permet de mettre à \perp les variables de requêtes de p_k et au fils de p_k de transmettre l'acquiescement à son fils sur l'arbre. Ainsi de suite jusqu'à ce que les règles \mathbf{R}_1 et \mathbf{C}_2 soient applicables. Après exécution de ces deux règles, la seule règle applicable sur p_1 est \mathbf{C}_1 et son application mettra à \perp les variables de requêtes de p_1 . Puisque p_i maintient une requête vers r par p_1 , $Asks(0, 1)$ est vrai. De même pour $Grants(1, 0)$ suite à l'exécution de \mathbf{R}_1 . Par conséquent, la seule règle applicable est \mathbf{J}_0 et son exécution permettra au processeur p_0 de rejoindre l'arbre avec $par_0 = 1$. ■

Lemme 5.26 *Soit r la plus grande identité du réseau. Dans tout 3-suffixe de l'exécution, à partir d'un certain état l'arbre couvre l'ensemble des processeurs.*

Preuve. Nous itérons le raisonnement du lemme 5.25 à tous les processeurs appartenant à $Arbre^c$. A partir d'un certain état, toutes les requêtes faites par ces processeurs seront acquittées par la racine. Une fois ses requêtes acquittées, chacun de ces processeurs rejoint l'arbre et ne le quitte plus. ■

5.7 Conclusion

Dans ce chapitre nous nous sommes intéressés à la propriété de confinement de fautes dans la résolution du problème de l'élection et de l'arbre couvrant. Cette propriété garantit qu'un algorithme limite les effets des fautes transitoires frappant un petit nombre de composants du système. Nous avons étudié l'algorithme de Afek et al. (1990), qui construit un arbre couvrant enraciné sur le processeur d'identité maximale dans le cadre d'un système asynchrone et non uniforme. A notre connaissance, ce type de système n'avait pas été étudié jusqu'à maintenant sous l'angle du confinement de fautes. Pourtant, ces caractéristiques peuvent se rencontrer facilement dans les réseaux à grande échelle. A titre d'exemple, dans des réseaux tel qu'Internet, il est désirable qu'une faute transitoire au niveau d'un routeur soit réparée le plus rapidement possible et n'affecte qu'un petit nombre de routeurs. Ainsi, la faute est transparente pour l'utilisateur du réseau.

Pour intégrer la propriété de confinement de faute à l'algorithme de Afek et al., nous l'avons modifié de façon à ce que seuls le site fautif et ses voisins soient susceptibles d'exécuter des actions permettant de reconstruire l'arbre suite à une panne singulière. Pour cela, sous l'hypothèse qu'un processeur peut lire les variables des processeurs voisins

de ses propres voisins, chaque processeur vérifie la cohérence de la structure locale de l'arbre à l'aide de prédicats. De plus, nous évitons qu'un des voisins du site fautif se rattache à celui-ci, ou qu'un de ses fils se détache de l'arbre. Après avoir décrit les règles de l'algorithme, nous prouvons qu'à partir d'un état incorrect obtenu à la suite d'une faute touchant un unique processeur, seuls ce dernier et ses voisins exécutent une règle. Cette exécution conduit soit à un état correct, soit à un état simplement correct. L'auto-stabilisation de l'algorithme est prouvée en montrant la disparition des fausses identités, le maintien d'une requête d'un processeur n'appartenant pas à l'arbre stabilisé vers un processeur de cet arbre, et la couverture par l'arbre de tous les processeurs du système.

Conclusion

Dans cette thèse, nous nous sommes efforcés, d'une part d'importer l'approche auto-stabilisante, habituellement utilisée dans des réseaux filaires, dans un contexte de réseau mobile, d'autre part d'intégrer la propriété de confinement de fautes dans un algorithme auto-stabilisant de construction d'un arbre couvrant. Pour ces deux problématiques, la spécificité de notre travail réside dans la prise en compte simultanée de deux préoccupations courantes dans les algorithmes répartis : adéquation à un contexte mobile et tolérance aux pannes, résolution d'un problème d'arbre couvrant et traitement local des fautes. Ces préoccupations n'étant pas nécessairement aisées à concilier, cela complique singulièrement la tâche de conception d'un algorithme réparti adéquat. Dans cette conclusion, nous effectuons une synthèse de notre contribution puis nous évoquons quelques perspectives de recherche qui nous semblent intéressantes.

Synthèse

L'intérêt principal des algorithmes auto-stabilisants est de fournir une tolérance naturelle aux défaillances transitoires. Le coût d'une telle propriété est l'existence d'une phase de stabilisation pendant laquelle les défaillances peuvent influencer le système, se propager et produire un comportement incorrect. Le comportement des algorithmes auto-stabilisants pendant cette phase de convergence est donc particulièrement étudié.

Dans la première partie de ce travail, nous avons étendu le domaine d'application de l'auto-stabilisation, jusque-là plus particulièrement réservée aux réseaux filaires, au systèmes de robots mobiles. Dans un premier temps, nous avons présenté un algorithme d'ordonnancement auto-stabilisant, basé sur une stratégie de visites des points de rendez-vous assurant qu'après la phase de stabilisation, chaque visite aboutit à une communication. La particularité de notre approche réside dans l'utilisation des réseaux de Petri comme modèle formel pour vérifier la correction de l'algorithme. Dans un deuxième temps, nous avons présenté un autre algorithme auto-stabilisant, basé sur le premier, qui construit une forêt d'arborescences des plus courts chemins des points de rendez-vous d'un robot donné à l'ensemble des points de rendez-vous des autres robots.

Pour palier aux limites de l'auto-stabilisation, des approches ont été développées pour un traitement plus local et plus rapide des fautes singulières, consistant à les circonscrire à une zone proche des processeurs corrompus et à contraindre le temps de stabilisation. L'une de ces approches, le confinement de fautes, fait l'objet de la deuxième partie de notre travail. Nous nous sommes efforcés d'introduire cette notion dans l'algorithme d'Afek, Kutten et Yung (1990) de construction d'arbre couvrant dans un système asynchrone et non uniforme. Le protocole proposé fonctionne sous l'hypothèse qu'un site puisse lire les variables locales des voisins de ses propres voisins.

Perspectives

Les perspectives d'évolution sont nombreuses pour les deux parties. Nous présentons ici quelques points sur lesquels nous avons espoir de travailler.

Auto-stabilisation dans les réseaux mobiles. Les systèmes autonomes mobiles sont particulièrement bien adaptés aux changements inhérents aux systèmes dynamiques. Ils fonctionnent de façon continue jusqu'à l'arrivée d'une perturbation qui déplace l'état du système. Pour la poursuite de notre étude, nous envisageons d'aborder les points suivants :

- *Etude de la faisabilité d'un algorithme déterministe de synchronisation de robots.* Nous conjecturons la possibilité de conception d'un algorithme déterministe auto-stabilisant d'ordonnancement des visites des points de rendez-vous. En effet, la seule source de non-déterminisme dans l'algorithme proposé au chapitre 3 est le tirage aléatoire réalisé au cours de l'exécution de l'action MISS. Ce tirage vise à parer aux situations de blocage total qui peuvent survenir suite à l'exécution de l'algorithme non auto-stabilisant à partir de certaines configurations initiales. Il s'agirait donc de résoudre ces situations sans faire appel à un tirage aléatoire.
- *Le développement des algorithmes adaptés aux variations de l'environnement,* comme par exemple la modification des ressources matérielles (e.g., ajout ou suppression de robots). Il s'agit de proposer des stratégies de rendez-vous prenant en compte les inaccessibilités des points de rendez-vous. En effet, chaque robot dispose d'une identité unique dans le système, utilisée dans l'algorithme de routage. Ainsi, l'ajout d'un nouveau robot pourrait violer l'unicité des identités. Pour éviter ce type de problème, il pourrait être intéressant de composer en parallèle un algorithme auto-stabilisant de nommage, de façon à assigner à chaque robot, y-compris le nouvel arrivant, une identité unique.
- *L'étude de la prise en compte de coûts de communication.* En effet, le graphe de communication pour la recherche du plus court chemin est construit à partir des or-

donnancements des déplacements des robots. Chaque point de rendez-vous partagé par deux robots est représenté par deux nœuds dans ce graphe. Ainsi, l'instantanéité supposée des communications entre robots n'est pas respectée. Deux voies de recherche se présentent donc : la prise en compte explicite de coûts de communication entre les robots en attribuant un coût à tout arc reliant deux nœuds correspondant à un même point de rendez-vous, ou alors l'étude d'un autre modèle graphique où l'on aurait plus qu'un seul nœud par point de rendez-vous.

Confinement de fautes dans le problème de l'élection. Notre principal objectif est de concevoir des algorithmes confinant les fautes pour la construction d'un arbre couvrant. Pour la poursuite de notre étude, nous envisageons d'aborder les points suivants :

- *La levée de l'hypothèse sur le périmètre de lecture.* L'une des hypothèses fortes que nous avons imposée est qu'un site peut lire les variables des voisins de ses propres voisins. Nous envisageons d'étudier la possibilité de lever celle-ci, et de supposer plutôt qu'un site ne peut lire que les variables de ses voisins. Il s'agirait alors de redéfinir l'état local d'un processeur en ajoutant des variables auxiliaires fournissant les informations nécessaires concernant les voisins des voisins. Ces variables étant elles aussi sujettes aux fautes transitoires, une synchronisation entre les voisins s'imposerait afin de vérifier la correction de ces variables et leur mise à jour.
- *La gestion de groupe dans un réseau à grande échelle.* L'avènement et l'utilisation croissante des réseaux à grande échelle de type WAN a contribué à l'émergence de nouveaux types d'applications réparties. L'abstraction de groupe (logique ou physique) s'est avérée très utile dans ces nouvelles applications, impliquant souvent des réseaux locaux (LANs) multiples. Etant donné notre double intérêt pour la communication de groupe et l'auto-stabilisation dans les systèmes répartis, notre démarche viserait à associer ces deux notions en concevant des protocoles auto-stabilisants hiérarchiques impliquant des groupes.

Notre objectif serait donc de permettre une communication de groupe à grande échelle pour assurer la construction d'un arbre couvrant dans un tel contexte. L'idée serait d'étudier la possibilité de proposer un algorithme auto-stabilisant de construction d'un arbre couvrant. Cet algorithme devra reposer sur les deux aspects suivants :

- chaque ensemble local de participants constituera un groupe. A un instant donné, un seul participant sera placé à la racine de l'arbre couvrant.
- chaque groupe sera structuré sur la base d'un arbre dont la racine sera le membre qui possédera le plus grand identificateur parmi tous les membres de son groupe. Ce membre demeurera racine jusqu'à ce que l'arbre inter-groupe soit construit.

De ce fait, les utilisateurs d'un même réseau local (LAN) constitueront les participants d'un groupe local. Un participant sera inclus dans un et un seul groupe à la fois. Autrement dit, il n'y aura pas d'élément commun entre les groupes. Chaque participant sera identifié par son adresse réseau qui sera son unique identifiant. Dans le cas d'un groupe, une adresse de diffusion lui sera attribuée, que nous considérerons comme l'unique identificateur du groupe. Nous distinguerons, dans cet algorithme, deux types d'arbres : l'arbre local correspondant au fonctionnement local au sein d'un groupe, et l'arbre global construit par les racines des arbres locaux. Au sein de chaque groupe, les participants exécuteront en parallèle deux algorithmes confinant les fautes. Le premier aura pour objet la création d'un arbre couvrant tous les participants du groupe, tandis que le deuxième visera à contruire un arbre global couvrant les racines des arbres locaux.

Dans un premier temps, nous pensons utiliser le même algorithme de construction d'arbre couvrant au niveau inter et intra-groupe. Nous pourrions ensuite envisager de composer deux algorithmes différents de construction d'un arbre couvrant, selon qu'on se situerait au niveau local ou global. Si nous y parvenons, on pourrait alors viser à introduire le confinement de fautes dans l'algorithme de niveau local.

A travers ce document, nous espérons avoir contribué au développement d'approches coopératives tolérantes aux pannes. En effet, l'occurrence de pannes étant quasiment inévitable dans les systèmes distribués du fait de la multiplicité des composants, l'aspect de tolérance aux pannes ne peut à notre sens être occulté dans la conception d'algorithmes répartis. C'est pourquoi il nous semble indispensable d'étendre les approches tolérantes aux fautes à la plus large gamme possible de problèmes.

Bibliographie

- Y. Afek et S. Dolev (1997) “Local stabilizer”, *Proceedings of the 5th Israeli Symposium on Theory of Computing and Systems*, pp. 74–84.
- Y. Afek, S. Kutten et M. Yung (1990) “Memory-efficient self-stabilization on general networks”, *Proceedings of the 4th International Workshop on Distributed Algorithms and Graphs, WADG'90*, pp. 15–28.
- S. Aggarwal et S. Kutten (1993) “Time optimal self-stabilizing spanning tree algorithm”, dans *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS93, Springer-Verlag LNCS :761*, pp. 400–410.
- D. Angluin (1980) “Local and global properties in networks of processors”, dans *Proceedings of the 12th annual ACM symposium on Theory of computing*, ACM Press, pp. 82–93.
- A. Arora et M. Gouda (1994) “Distributed reset”, *IEEE Transactions on Computers*, tm. 43, no. 9, pp. 1026–1038.
- A. Arora, M. Gouda et T. Herman (1990) “Composite routing protocols”, dans *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pp. 70–78.
- A. Arora et H. Zhang (2003) “Lsrp : Local stabilization in shortest path routing”, *International Conference on Dependable Systems and Networks, DSN 2003*.
- H. Attiya et J. Welch (1998) *Distributed Computing : Fundamentals, Simulations, and Advanced Topics, 2nd Edition*, John Wiley and Sons, 1^{re} édn.
- H. Baala, O. Flauzac, J. Gaber, M. Bui et T. El Ghazawi (2003) “A self-stabilizing distributed algorithm for spanning tree construction in wireless ad hoc networks”, *Journal of Parallel and Distributed Computing*, tm. 63, no. 1, pp. 97–104.
- J. Beauquier, B. Bérard, L. Fribourg et F. Magniette (2001) “Proving convergence of self-stabilizing systems using first-order rewriting and regular languages”, *Distributed Computing*, tm. 14, no. 2, pp. 83–95.

- J. Beauquier, C. Genolini et S. Kutten (1998) “k-stabilization of reactive tasks”, *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing, PODC’98*, p. 318.
- J. Beauquier, C. Genolini et S. Kutten (1999a) “Optimal reactive k-stabilization : the case of mutual exclusion”, *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing, PODC’99*, pp. 209–218.
- J. Beauquier, M. Gradinariu et C. Johnen (1999b) “Memory space requirements for self-stabilizing leader election protocols”, dans *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pp. 199–208.
- J. Beauquier, M. Gradinariu et C. Johnen (1999c) “Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings”, *Rapport Technique 1225, Université Paris Sud*.
- P. Bracka, S. Midonnet et G. Roussel (2002) “Routage dans un réseau de robots”, *Quatrièmes Rencontres Francophones sur les aspects Algorithmiques des Télécommunications AlgoTel*, pp. 17–24.
- R. Brooks et A. Flynn (1989) “Fast, cheap and out of control : a robot invasion of the solar system”, *Rap. tech. 1182*, AIM, URL citeseer.ist.psu.edu/brooks89fast.html.
- A. Bui, A. K. Datta, F. Petit et V. Villain (1999a) “Snap-stabilizing PIF algorithm in tree networks without sense of direction”, dans *Proceedings of the 6th International Colloquium on Structural Information and Communication Complexity, SIROCCO’99*, Carleton University Press, pp. 32–46.
- A. Bui, A. K. Datta, F. Petit et V. Villain (1999b) “State optimal and snap-stabilizing pif on tree networks”, dans *ICDCS 1999, Workshop on Self-Stabilizing Systems (WSS’99)*, IEEE Computer Society Press, pp. 78–85.
- J. E. Burns et J. K. Pachl (1989) “Uniform self-stabilizing rings”, *ACM Transactions on Programming Languages and Systems*, tm. 11, no. 2, pp. 330–344.
- Y. U. Cao, A. Fukunaga et A. Kahng (1997) “Cooperative mobile robotics : Antecedents and directions”, *Autonomous Robots*, tm. 4, pp. 7–23.
- N. S. Chen, H. P. Yu et S. T. Huang (1991) “A self-stabilizing algorithm for constructing spanning trees”, *Information processing Letters*, tm. 39, no. 3, pp. 147–151.
- Y. Chen, A. Datta et S. Tixeuil (2002) “Stabilizing inter-domain routing in the internet”, *Lecture Notes in Computer Science 2400, In Euro-Par’02 Parallel Processing*, pp. 749–752.

- J. A. Cobb et M. G. Gouda (2002) “Stabilization of general loop-free routing”, *Journal of Parallel and Distributed Computing*, tm. 62, no. 5, pp. 922–944.
- A. K. Datta, J. L. Derby, J. E. Lawrence et S. Tixeuil (2000) “Stabilizing hierarchical routing”, *Journal of Interconnection Networks*, tm. 1, no. 14, pp. 283–302.
- X. Defago et A. Konagaya (2002) “Circle formation for oblivious anonymous mobile robots with no common sense of orientation”, dans *Workshop on self stabilizing, POMC’02*, pp. 97–104.
- M. Demirbas, A. Arora et M. Gouda (2003) “A pursuer-evader game for sensor networks”, *Sixth Symposium on Self-Stabilizing Systems, SSS’03*, pp. 1–16.
- M. Diaz (2001) *Les réseaux de Petri*, Informatique et Système d’Information, Hermès.
- E. Dijkstra (1974) “Self-stabilizing systems in spite of distributed control”, *Communications of the ACM*, tm. 17, no. 11, pp. 643–644.
- S. Dolev (1997) “Self-stabilizing routing and related protocols”, *Journal of Parallel and Distributed Computing*, tm. 42, no. 2, pp. 122–127.
- S. Dolev (2000) *Self-stabilization*, MIT Press.
- S. Dolev et T. Herman (1995) “Superstabilizing protocols for dynamic distributed systems”, *Proceedings of the 2nd Workshop on Self-stabilizing Systems*, pp. 3.1–3.15.
- S. Dolev et T. Herman (1999) “Parallel composition of stabilizing algorithms”, dans *WSS’99*, pp. 25–32.
- S. Dolev, A. Israeli et S. Moran (1989) “Self-stabilization of dynamic systems”, *Proceedings of the MCC Workshop on Self-stabilizing systems*, tm. MCC Technical Report No. STP-379-89.
- S. Dolev, A. Israeli et S. Moran (1993) “Self-stabilizing of dynamic systems assuming only read/write atomicity”, *Distributed Computing*, tm. 7, pp. 3–16.
- S. Dolev, A. Israeli et S. Moran (1997a) “Resource bounds for self-stabilization message driven protocols”, *SIAM on Computing*, pp. 2273–290.
- S. Dolev, A. Israeli et S. Moran (1997b) “Uniform dynamic self-stabilizing leader election”, *IEEE Transactions on Parallel and Distributed Systems*, tm. 8, no. 4, pp. 424–440.
- A. Drogoul et D. Fresneau (1998) “Métaphore de fourragement et modèle d’exploitation collective de l’espace pour des colonies de robots autonomes mobiles”, *Actes des Journées Francophones IAD et SMA, JFIADSMA ’98*, pp. 16–32.

- M. Dufflot (2003) *Algorithmes distribués sur des anneaux paramétrés - Preuves de convergence probabiliste et déterministe*, Thèse de doctorat, Ph.D. Thesis, Laboratoire Spécification et Vérification, ENS de Cachan.
- J. El Haddad et S. Haddad (2003a) “Algorithmes de communication auto-stabilisants dans un système de robots mobiles”, dans *Colloque Francophone sur la Modélisation des Systèmes Réactifs, MSR’03*, Metz, France.
- J. El Haddad et S. Haddad (2003b) “Self-stabilizing scheduling algorithm for cooperating robots”, dans *ACS/IEEE International Conference on Computer Systems and Applications, AICCSA’03*, Tunis, Tunisie.
- J. El Haddad et S. Haddad (2004a) “A fault-contained spanning tree protocol for arbitrary networks”, dans *Proceedings of the 17th International Conference on Parallel and Distributed Computing Systems, PDCS’04*, San Francisco, USA.
- J. El Haddad et S. Haddad (2004b) “A fault-tolerant communication mechanism for cooperative robot”, *International Journal of Production Research*, tm. 42, no. 14, pp. 2793–2808.
- W. Feller (1968) *An introduction to probability theory and its applications*, tm. I, John Wiley & Sons, Paris, third edition.
- M. Fischer, N. Lynch et M. Paterson (1985) “Impossibility of distributed consensus with one faulty process”, *Journal of the Association of the Computing Machinery*, tm. 32, no. 2, pp. 374–382.
- C. Fiévet (2004) “Bienvenue dans le monde des robots - en réseau”, article publié dans la lettre Internet Actu nouvelle génération, INIST-CNRS et FING, URL <http://fing.org/index.php?num=4832,4>.
- P. Flocchini, G. Prencipe, N. Santoro et P. Widmayer (1999) “Hard tasks for weak robots : The role of common knowledge in pattern formation by autonomous mobile robots”, dans *Proceedings of the 10th International Symposium on Algorithms and Computation*, Springer-Verlag, pp. 93–102.
- F. Gärtner (2003) “A survey of self-stabilizing spanning-tree construction algorithms”, *Rapport Technique 200338, l’Ecole Polytechnique Fédérale de Lausanne*.
- S. Ghosh et A. Gupta (1996) “An exercise in fault-containment : self-stabilization leader election”, *Information processing Letters*, tm. 59, pp. 281–288.

- S. Ghosh, A. Gupta, T. Herman et S. V. Pemmaraju (1996a) "Fault-containing self-stabilizing algorithms", In *Proceedings of the Fifteenth Annual ACM Symposium on Principales of Distributed Computing, PODC'96*, pp. 45–54.
- S. Ghosh, A. Gupta et S. V. Pemmaraju (1996b) "A fault-containing self-stabilizing algorithm for spanning trees", *Journal of Computing and Information*, tm. 2, pp. 322–338.
- S. Ghosh, A. Gupta et S. V. Pemmaraju (1997) "Fault-containing network protocols", In *Proceedings of the 12th Annual ACM Symposium on Applied Computing*.
- S. Ghosh et X. He (2000) "Fault-containing self-stabilization using priority scheduling", *Information Processing Letters*, tm. 73, pp. 145–151.
- S. Ghosh et X. He (2002) "Scalable self-stabilization", *Journal of Parallel and Distributed Computing*, tm. 62, pp. 945–960.
- M. G. Gouda (1995) "The triumph and tribulation of system stabilization", *Lecture Notes in Computer Science, Springer Verlag*, tm. 972, pp. 1–18.
- M. G. Gouda et T. Herman (1991) "Adaptative programming", *IEEE TSE*, tm. 17, pp. 911–921.
- M. G. Gouda et N. J. Multari (1991) "Stabilizing communication protocols", *IEEE Transactions on Computers*, tm. 40, no. 4, pp. 448–458.
- M. G. Gouda et M. Schneider (1999) "Stabilization of maximal metric trees", dans *Proceedings of the fourth Workshop on Self-stabilizing Systems, WSS99*, IEEE Computer Society, pp. 10–17.
- M. G. Gouda et M. Schneider (2003) "Maximizable routing metrics", *IEEE/ACM Transactions on Networking*, tm. 11, no. 4, pp. 663–675.
- M. Grandinariu (2000) *Modélisation, vérification et raffinement des algorithmes auto-stabilisants*, Thèse de doctorat, Ph.D. Thesis, Laboratoire de Recherche en informatique, Université Paris XI, Orsay.
- A. Gupta (1997) *Fault-containment in self-stabilizing distributed systems*, Thèse de doctorat, University of Iowa, USA.
- K. P. Hatzis, G. P. Pentaris, P. G. Spirakis, V. T. Tampakas et R. B. Tan (1999) "Fundamental control algorithms in mobile networks", dans *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, pp. 251–260.

- T. Herault (2003) *Correction de défaillances transitoires dans les systèmes auto-stabilisants*, Thèse de doctorat, Ph.D. Thesis, Laboratoire de Recherche en informatique, Université Paris XI, Orsay.
- T. Herman (1991) *Adaptativity through distributed convergence*, Thèse de doctorat, Ph.D. Thesis, University of Texas at Austin, Departement of Computer Science.
- T. Herman (2000) “Superstabilizing mutual exclusion”, *Distributed Computing*, tm. 13, pp. 1–17.
- H. Hu, I. Kelly, D. Keating et D. Vinagre (1998) “Coordination of multiple mobile robots via communication”, dans *Proceedings of SPIE. Mobile Robots XIII and Intelligent Transportation Systems*, Boston, Massachusetts, pp. 94–103.
- S. Huang et N. S. Chen (1992) “A self-stabilizing algorithm for constructing breadth-first trees”, *Information processing Letters*, tm. 41, pp. 109–117.
- C. Johnen, F. Petit et S. Tixeuil (2004) “Auto-stabilisation et protocoles réseau”, *Rapport de Recherche 1357, Université Paris Sud, Technique et Science Informatique, à paraître*.
- J. Kemeny et J. Snell (1960) *Finite Markov Chains*, Van Nostrand, Princeton, NJ.
- S. Kutten et B. Patt-Shamir (1997) “Time-adaptive self stabilization”, *Symposium on Principles of Distributed Computing*, pp. 149–158.
- S. Kutten et D. Peleg (1999) “Fault-local distributed mending”, *Information processing Letters*, tm. 30, pp. 144–165.
- L. Lamport (1978) “Time, clocks and the ordering of events in a distributed system”, *Communications of the ACM*, tm. 21, pp. 558–565.
- L. Lamport (1984) “Solved problems, unsolved problem and non-problems in concurrency, Invited Address.”, *Processdings of the Third ACM JSymposium on Principales of Distributed Computing*, pp. 1–11.
- L. Lamport, R. Shostak et M. Pease (1982) “The byzantine generals problem”, *ACM Transactions on Programming Languages and Systems*, tm. 4, pp. 382–401.
- I. Lavallé (1990) *Algorithmiques parallèle et distribué.*, Traité des Nouvelles Technologies, serie Informatique. Hermès.
- X. Lin et S. Ghosh (1991) “Self-stabilizing maxima finding”, *Processdings of the 28th Annual Allerton Conference*, pp. 662–671.

- D. Luzeaux (2000) “Autonomous small robots for military applications”, dans *Conference on Unmanned Ground Vehicle Technology*, USA.
- N. Lynch (1996) *Distributed Algorithms*, Morgan Kaufmann.
- F. Magniette (2002) *Preuve d’algorithmes auto-stabilisants*, Thèse de doctorat, Ph.D. Thesis, Laboratoire de Recherche en informatique, Université Paris XI, Orsay.
- M. Nesterenko et A. Arora (2001) “Self-stabilizing routing in wireless embedded systems”, dans *SRDS 2001, Workshop on Reliability in Embedded Systems*, pp. 16–22.
- V. Park et M. Corson (1997) “A highly adaptive distributed routing algorithm for mobile wireless networks”, dans *Proceedings IEEE INFOCOM, The Conference on Computer Communications, Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, tm. 3, pp. 1405–1413.
- L. Parker (1998) “Alliance : An architecture for fault-tolerant multi-robot cooperation”, *IEEE Transactions on Robotics and Automation*, tm. 14, pp. 220–240.
- F. Petit (1998) *Efficacité et simplicité dans les algorithmes distribués auto-stabilisants de parcours en profondeur de jeton*, Thèse de doctorat, Ph.D. Thesis, Technical Report LaRIA-98-05.
- G. Prencipe (2001) “Corda : Distributed coordination of a set of autonomous mobile robots”, dans *European Research Seminar on Advances in Distributed Systems, Ersads*.
- M. Raynal (1991) *La communication et le temps dans les réseaux et les systèmes répartis*, Collection de la DER d’EDF. Eyrolles, Tome 1 d’une introduction aux principes des systèmes répartis.
- M. Raynal (1992a) *Gestion des Données Répartis : Problèmes et Protocoles.*, Eyrolles.
- M. Raynal (1992b) *Synchronisation et état global dans les systèmes répartis*, Collection de la DER d’EDF. Eyrolles, Tome 2 d’une introduction aux principes des systèmes répartis.
- W. Reisig (1985) *Petri Nets : an Introduction*, Springer Verlag.
- M. Schneider (1993) “Self-stabilization”, *ACM Symposium Computing Surveys*, tm. 25, pp. 45–67.
- P. Sellem et D. Luzeaux (2000) “Répartition de la perception dans un système distribué de robots autonomes”, dans *Journées Francophones d’Intelligence Artificielle Distribuée et Système multi-agents*, France.

- I. Suzuki et M. Yamashita (1999) “Distributed anonymous mobile robots : Formation of geometric patterns”, *SIAM Journal on Computing*, tm. 28, no. 4, pp. 1347–1363.
- G. Tel (1994) *Introduction to Distributed Algorithms*, Cambridge University Press, 1^{re} édn.
- S. Tixeuil (2000) *Auto-stabilisation Efficace*, Thèse de doctorat, Laboratoire de Recherche en Informatique, Université Paris XI, Orsay, France.
- G. Varghese (1997) “Compositional proofs of self-stabilizing protocols”, *Proceedings of the Third Workshop on Self-stabilizing Systems*, pp. 80–94.
- B. Werber (1991) *Les fourmis*, Albin Michel.
- B. Werber (1995) *La révolution des fourmis*, Albin Michel.
- H. Zhang et A. Arora (2002) “GS³ : Scalable self-configuration and self-healing in wireless networks”, dans *Proceedings of the twenty-first annual symposium on Principles of distributed computing, PODC 2002*, pp. 58–67.
- W. Zhang, Z. Deng, G. Wang, L. Wittenburg et Z. Xing (2002) “Distributed problem solving in sensor networks”, dans *Proceedings of the first international joint conference on Autonomous agents and multiagent systems, AAMAS-2002*, pp. 988–989.

Vu :
Le Président
M.

Vu :
Les Suffrageants
M.

Vu et permis d'imprimer :
Le Vice-Président du Conseil Scientifique Chargé de la Recherche de l'Université Paris
IX Dauphine

Résumé

Cette thèse porte sur l'étude des algorithmes répartis tolérant aux pannes. Plusieurs approches ont été proposées dans cette optique, parmi lesquelles l'auto-stabilisation (le système rétablit de lui-même un fonctionnement correct au bout d'un temps fini) et le confinement de fautes (on limite la propagation d'une faute singulière). Nos travaux visent à étendre les champs d'application de ces deux approches. Concernant l'approche auto-stabilisante, nous proposons un algorithme d'ordonnancement des déplacements dans un système de robots mobiles communiquant sur des points de rendez-vous fixes. Cet algorithme garantit que, après la phase de stabilisation, chaque visite à un point de rendez-vous aboutit à une communication. Nous présentons ensuite un algorithme auto-stabilisant pour engendrer et maintenir les tables de routage des robots. Concernant le confinement de fautes, nous présentons un algorithme qui résout le problème de l'élection et de l'arbre couvrant dans un graphe quelconque.

Mots-clés : Algorithmique répartie, tolérance aux pannes, auto-stabilisation, confinement de fautes.

Abstract

This thesis deals with fault tolerant distributed algorithms. Several approaches were developed for this purpose, such as self stabilization (algorithms that guarantee a return to correct behavior within finite time after all faulty behavior ceases) and fault containment (algorithms that contain the effects of limited transient faults). Our work aims at extending the application of both of these approaches. Concerning self stabilization, we designed a scheduling protocol for a network of robots communicating on a fixed set of locations. This algorithm solves the management of visits to these locations ensuring that after the stabilizing phase, every visit to a location will lead to a communication. Another algorithm computes the minimum hop path between a specific robot's location and the locations of all the other robots of the system in order to implement routing. Concerning fault containment, we design an algorithm for constructing a rooted spanning tree in an arbitrary network.

Keywords : Distributed algorithms, fault tolerance, self stabilization, fault containment.