Quelques Contributions à la Stabilisation Instantanée

Stéphane Devismes

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE PICARDIE JULES VERNE

— AMIENS—

POUR L'OBTENTION DU GRADE DE

DOCTEUR

Spécialité : Informatique

Par STÉPHANE DEVISMES

Quelques Contributions à la Stabilisation Instantanée

Soutenue publiquement le 8 décembre 2006 devant le jury composé de :

M.	PATRICE SÉÉBOLD ...	Professeur	Président
M.	YVES MÉTIVIER	Professeur	Rapporteur
M.	SÉBASTIEN TIXEUIL .	Maître de Conférences (HDR)	Rapporteur
M.	HUGUES FAUCONNIER	Maître de Conférences (HDR)	Examinateur
M.	FRANCK PETIT.....	Professeur	Examinateur
M.	ALAIN COURNIER....	Maître de Conférences	Directeur
M.	VINCENT VILLAIN ...	Professeur	Directeur

Laboratoire de Recherche d'Informatique d'Amiens
CNRS FRE 2733, Université de Picardie Jules Verne

Rapport de Recherche N ° 2006-09

Soyez réalistes : demandez l'impossible.
Che Guevara

Remerciements

J’ai eu l’honneur de travailler avec **Alain Cournier** et **Vincent Villain** à partir de mon stage de DEA. Durant ce stage ainsi que pendant toute la durée de ma thèse, j’ai pu apprécier leurs nombreuses qualités tant professionnelles que personnelles. J’ai beaucoup appris à leurs côtés notamment au cours de nos nombreux débats “philosophiques” : le goût de la recherche, le besoin de se remettre en question, la nécessité d’être extrêment rigoureux ... Je les remercie tous les deux pour leur gentillesse et leur disponibilité au cours de ces trois dernières années. Je leur fais part de ma sincère reconnaissance pour la grande patience qu’ils ont eue avec moi.

Je remercie **Patrice Séébold** d’avoir accepté de participer à mon jury. Il reste pour moi l’un des meilleurs enseignants (si ce n’est le meilleur) que j’ai pu rencontrer au cours de mes études. À ce titre, il fut l’un de ceux, avec Alain et Vincent, qui m’a donné envie de faire de la recherche. En tout cas, je considère Patrice comme un modèle à suivre en matière d’enseignement.

J’ai côtoyé **Franck Petit** depuis ma licence et j’ai eu le plaisir de collaborer avec lui pour l’un de mes premiers articles. À cette occasion, j’ai pu apprécier ses nombreuses qualités humaines et scientifiques. Je le remercie également de participer à mon jury.

J’ai rencontré **Sébastien Tixeuil** lors de ma première conférence. Depuis, nous nous sommes retrouvés à de nombreuses reprises. J’ai pu apprecier sa simplicité et sa gentillesse à chacune de nos rencontres. De plus, j’ai le plus grand respect pour son travail scientifique. C’est donc tout naturellement que je lui ai demandé d’être l’un de mes rapporteurs. Nos échanges au cours de la phase de relecture de cette thèse ont, je pense, permis d’améliorer significativement la qualité de ce mémoire. Je le remercie pour cela ainsi que pour son amitié.

C’est un honneur pour moi qu’**Yves Métivier** ait accepté de rapporter ma thèse. Je le remercie de l’intérêt qu’il a porté à mes travaux.

C’est aussi une grande satisfaction qu’**Hugues Fauconnier** participe à mon jury. Je le remercie également chaleureusement pour son accueil lors de mon arrivée au LIAFA. Je souhaite que la collaboration que nous venons d’amorcer soit fructueuse. Je joins également **Carole Delporte** à ces remerciements. Merci encore à vous deux.

Je remercie ma famille et mes amis pour leur soutien continu durant ces trois années. Merci à **Émilie** pour sa patience infinie. Merci d’avoir pardonné mes nombreuses absences aussi bien physiques que mentales.

Merci à ma mère, **Annick**, pour les valeurs qu’elle m’a inculquées. Merci pour son soutien moral et financier. Merci à mon frère **Christophe** et ma soeur **Delphine**.

Merci aux amis : **Charlie**, **Sébastien et Céline**, **Julien**, **Alex et Mary**, **Stéphane**, **Guillaume et Sandrine**, **Guillaume et Anissa**. Merci d’avoir supporté ma mauvaise humeur chronique !

Merci aux amis doctorants : **Sidney, Sylvain, Gary, Hakim, Yoann, Hafedh, Cheikh, Sabine, Djibo, Christian et les autres**. Vous êtes très vite passés de collègues à amis. Les pauses clopes et la petite mousse de 17 heures, ça rapproche ! J’ai déjà la nostalgie de nos discussions sur la recherche, le cinéma et la musique...

Merci aux amis anciens doctorants : **Sandra, Olivier, Bernard et Florence**. Leur aide et leur expérience m’ont été précieuses.

Merci aux permanents du LaRIA. En particulier, les duettistes **Laure et Gilles** ainsi que le directeur de laboratoire **Gilles Kassel**.

Merci à **Sylvie** pour le café dès 9 heures !

Merci à **Christophe** pour le réseau mais pas pour le café, il est vraiment trop fort !

Merci à **Ajoy** de m’avoir appris l’anglais !

Merci au **Judo Club Abbevillois**, en particulier, **Michel, Dany et Benoit**. 18 années de ma vie qui m’ont forgé le caractère.

Table des matières

Index

Chapitre 1

Introduction

Les premiers réseaux informatiques sont apparus au début des années 50 pour des applications militaires ([Kra87]). Par exemple, le projet SAGE conçu par IBM pour l'US Air Force consistait à coordonner un flux de messages depuis les radars jusqu'aux unités d'interception, permettant ainsi de réduire le temps requis pour contrer les attaques éventuelles de bombardiers. En ce qui concerne les premières applications civiles nous pouvons citer le système de réservation de vols d'American Airlines : SABRE, qui fut déployé dans les années 60. Précédemment, le système de réservation était géré manuellement par une équipe de 8 opérateurs ! Avec une équipe aussi réduite, le temps de réservation restait acceptable tant que le nombre de vols restait limité. Ainsi, l'explosion du trafic aérien a eu raison d'un tel système.

Les premiers réseaux étaient dédiés à des applications bien déterminées où l'échange de données se faisait point à point sous le contrôle de l'application. À la fin des années 60 furent lancés les premiers projets de réseaux généraux à grande distance. L'un de ces précurseurs fut ARPANET (*Advanced Research Agency Network*). Ce projet fut lancé en 1967 par des universitaires américains et fut opérationnel deux ans plus tard. ARPANET servit de banc d'essai à de nouvelles technologies de gestion de réseau. Notamment, il fut le premier réseau à transmission par paquets. Le réseau *Internet* s'est largement inspiré du projet ARPANET. Internet a permis de rendre accessible de nombreux services à une échelle mondiale (Courriel, commerce en ligne, blogues, etc.).

Le boom d'Internet est révélateur de l'importance croissante des réseaux dans la plupart des secteurs d'activités. Les progrès des moyens de communication permettent aux réseaux à grande échelle tels qu'Internet d'échanger des volumes de données toujours plus importants. Cependant, la probabilité d'apparition d'une faute dans un réseau croît en fonction du nombre de machines inter-connectées. En conséquence, les réseaux à grande échelle sont naturellement sujets aux fautes. L'algorithmique distribuée s'est donc orientée depuis maintenant plusieurs années vers la prise en compte des défaillances potentielles des composants du réseau. Deux approches, *a priori* opposées, sont classiquement utilisées pour traiter ces défaillances :

- L'*algorithmique robuste* qui garantit qu'un système continue à se comporter correctement en dépit de fautes ;
- L'*algorithmique stabilisante* qui assure qu'après une perturbation temporaire, le système retrouve de lui-même et en un temps fini le comportement désiré.

L'algorithmique tolérante aux fautes a été bouleversée en 1985 par le résultat d'impossibilité de Fischer, Lynch et Paterson [FLP85]. Ce résultat montre que le problème du consensus n'admet pas de solution déterministe dans un système distribué totalement asynchrone (*i.e.*, chaque composant a une vitesse qui lui est propre, qui est inconnue des autres composants et qui n'a pas de borne) si une panne définitive est susceptible de se produire sur un processeur. Cette preuve d'impossibilité est fondée sur le fait que dans un système asynchrone, les voisins d'un processeur lent ne peuvent pas décider si ce processeur est effectivement lent ou s'il est en panne. Ce résultat important induit que sans

hypothèse supplémentaire, le problème du consensus ne peut pas être résolu de manière déterministe et robuste dans un système asynchrone. Ce résultat n'a pourtant pas mis fin à l'algorithmique robuste. De nouveaux résultats ont été obtenus avec des hypothèses supplémentaires. Par exemple, Chandra et Toueg [CT91] ont introduit la notion de *détecteur de panne*. Cette approche permet notamment de résoudre le problème du consensus en émettant certaines hypothèses sur la qualité du détecteur de panne utilisé.

Le résultat de Fischer, Lynch et Paterson est à l'origine du regain d'intérêt pour l'approche *auto-stabilisante* introduite par Dijkstra en 1974 [Dij74]. Cette approche permet de traiter les fautes transitoires. En effet, un système auto-stabilisant ne nécessite aucun état initial particulier pour converger en un temps fini vers un état à partir duquel il vérifie toujours les spécifications pour lesquelles il a été défini. Donc, un tel système recouvre naturellement (*i.e.* sans intervention extérieure) un comportement correct après que des fautes transitoires aient modifié les valeurs des variables et/ou le contenu des canaux de communication. Dans cette approche, les seules restrictions considérées sont que les fautes sont rares (pour que le système ait le temps de converger entre deux périodes de fautes) et qu'elles n'altèrent pas le code des protocoles (cette deuxième restriction peut être contournée en imposant un rafraîchissement périodique des codes). Le principal inconvénient de l'approche auto-stabilisante est que la perte de sûreté dûe aux fautes perdure pendant le temps de convergence du système. En 1999, une nouvelle approche appelée *stabilisation instantanée* a permis de résoudre ce problème ([BDPV99c]). En effet, la stabilisation instantanée permet de limiter l'absence de sûreté à la période (*a priori* incontrôlable) correspondant à l'existence de fautes transitoires dans le système, ainsi, le système vérifie ses spécifications quelle que soit la configuration dans laquelle il se retrouve après des fautes.

Bui, Datta, Petit et Villain ont défini un système instantanément stabilisant comme étant un système qui vérifie toujours ses spécifications quel que soit son état initial ([BDPV99c]). Bien sûr, dans un système instantanément stabilisant, l'état initial ainsi qu'un certain nombre des états suivants peuvent ne pas correspondre aux états générés par le système s'il avait été "proprement" initialisé. Cependant, contrairement aux systèmes auto-stabilisants, ces états "non souhaités" n'empêchent pas le système de vérifier sa spécification. Depuis l'article fondateur de Bui, Datta, Petit et Villain ([BDPV99c]), de nombreux protocoles instantanément stabilisants ont été proposés [PV03, CDPV02, BCV03, Nol02]. En outre, en 2003, Cournier, Datta, Petit et Villain ont montré que dans le modèle à états (*i.e.*, un modèle de calcul théorique où l'envoi de messages est simulé par des accès directs en lecture dans la mémoire des processeurs voisins) la stabilisation instantanée avait au moins la même puissance d'expression que l'auto-stabilisation, c'est-à-dire que, dans ce modèle, tout problème ayant une solution auto-stabilisante peut être résolu de manière instantanément stabilisante ([CDPV03]). Il faut noter que dans le modèle à états, la stabilisation instantanée est même plus expressive. En effet, des problèmes tels que la détection de terminaison qui ne peuvent pas être résolus en auto-stabilisation peuvent être résolus en stabilisation instantanée.

Dans cette thèse, nous nous sommes plus particulièrement intéressés au concept de stabilisation instantanée. Ainsi, nous allons tout d'abord proposer deux solutions instantanément stabilisantes au problème classique de *parcours en profondeur* distribué (aussi appelé *circulation de jeton*) pour des réseaux enracinés quelconques (chapitre 5). Le parcours en profondeur a de nombreuses applications. Dans les systèmes distribués, il est principalement utilisé pour résoudre le problème d'exclusion mutuelle. Mais, il peut aussi être utilisé pour résoudre, par exemple, le calcul d'arbre couvrant ([HC93]), la programmation par contrainte ([JSYZ04]), le routage par intervalle ([vLT87]) ou pour évaluer des propriétés globales sur le réseau (cf. [Tar72] et section 5.7). Les deux protocoles de parcours en profondeur instantanément stabilisants que nous proposons sont écrits dans le modèle à états. Le premier est basé sur des listes d'identités. Le second utilise un principe de question/réponse pour remplacer les listes d'identités. Il faut noter que ces deux solutions fonctionnent sous l'hypothèse d'un démon distribué inéquitable : le démon le plus général du modèle à états. Nous proposerons ensuite deux appli-

cations instantanément stabilisantes pouvant être obtenues à partir de nos deux protocoles de parcours en profondeur (chapitre 5). Ces deux applications évaluent des propriétés globales sur le réseau. La première application est un calcul de point fixe avec détection de terminaison. L'algorithme présenté permet de marquer les *points d'articulation* et les *isthmes* du réseau. Les points d'articulation (resp. les isthmes) sont des processeurs (resp. des canaux) dont la suppression (suite à une panne définitive, par exemple) provoque la partition du réseau survivant en plusieurs composantes connexes. De plus, l'existence de points d'articulation ou d'isthmes dans le réseau peut être l'une des causes d'apparition de congestions. L'identification des points d'articulation et des isthmes est donc essentielle du point de vue de la tolérance aux fautes. La seconde application permet d'évaluer si un ensemble donné est un *ensemble séparateur* du réseau. Un ensemble séparateur (*cutset*) est un sous-ensemble de processeurs du réseau dont la suppression (suite à des pannes définitives, par exemple) provoque la partition du réseau en plusieurs composantes connexes. La détection d'ensembles séparateurs est un problème important dans de nombreuses applications telles que l'évaluation de la fiabilité des réseaux.

Dans la seconde partie de nos résultats (chapitre 6), nous adoptons une approche plus générale qui consiste à étudier un protocole efficace de transformation semi-automatique de protocoles de service mono-initiateurs en protocoles instantanément stabilisants. En particulier, ce transformateur ne nécessite pas de calculs d'état global du système. Un protocole de parcours en profondeur et un protocole de construction d'arbre en largeur illustreront la facilité avec laquelle nous pouvons rendre instantanément stabilisants ces protocoles grâce à notre transformateur. Le protocole de parcours en profondeur est non seulement trivial à écrire mais les performances obtenues en font un compromis quasi idéal entre les protocoles à listes et à questions présentés dans le chapitre 5. Enfin, grâce à une propriété de comptage due à notre transformateur, nous montrerons comment utiliser ce protocole de parcours pour résoudre en quelques lignes l'exclusion mutuelle de manière instantanément stabilisante.

Organisation de la thèse

Cette thèse est organisée de la manière suivante :

Partie I : Cette partie est consacrée à la présentation des systèmes distribués.

> **Chapitre 2 :** Nous introduisons les principales caractéristiques des systèmes distribués ainsi que les problèmes classiques qui en découlent.
>
> **Chapitre 3 :** Nous nous consacrons plus précisément au problème de la tolérance aux fautes dans les systèmes distribués. En particulier, nous présentons les différentes approches algorithmiques de la littérature permettant à un système de tolérer les fautes.
>
> **Chapitre 4 :** Nous introduisons le modèle à états, *i.e.*, le modèle de calcul théorique que nous utilisons dans nos travaux.

Partie II : Dans cette seconde partie, nous présentons nos principaux résultats.

> **Chapitre 5 :** Nous proposons deux protocoles instantanément stabilisants de parcours en profondeur ainsi que deux applications dérivées de ces parcours.
>
> **Chapitre 6 :** Nous présentons un transformateur pour protocoles de service mono-initiateurs ainsi que trois extensions obtenues avec ce transformateur.

Le dernier chapitre — **Chapitre 7** — est consacré à la conclusion et aux perspectives.

Première partie

Problématique

Chapitre 2

Les Systèmes distribués

Dans ce chapitre, nous allons, tout d'abord, présenter quelques généralités sur les systèmes distribués (section 2.1). Puis, nous exposerons des caractéristiques propres à ces systèmes (section 2.2). Enfin, nous présenterons quelques problèmes fondamentaux des systèmes distribués (section 2.3).

2.1 Généralités sur les systèmes distribués

En informatique, un *système distribué* [Tel01] (aussi appelé *système réparti* ou *réseau*) est un ensemble d'unités de traitement autonomes interconnectées entre-elles. Ces unités de traitement, aussi appelées *processeurs*, coopèrent via des échanges d'*informations* (aussi appelés *messages*) dans le but de réaliser une tâche commune. Les informations transitent via des *liens de communications bidirectionnels*[1] (aussi appelés *canaux*).

Les deux principales contraintes liées aux systèmes distribués sont :

- *Le manque de connaissances globales sur le système*. Dans les hypothèses les plus faibles, les seules informations qu'a un processeur sur le réseau se résument à des étiquettes lui permettant de distinguer ses canaux.
- *L'absence de temps global*. Ce problème est dû, d'une part, à l'asynchronisme des horloges locales des processeurs ainsi qu'à leurs vitesses relatives et, d'autre part, au temps d'acheminement des messages.

La problématique majeure des systèmes distribués est donc de décrire formellement les mécanismes permettant de résoudre une tâche dont les données sont réparties dans le réseau tout en tenant compte de ces contraintes. Nous appelons *algorithme distribué* ou *protocole* la description des traitements (*i.e.*, envois, réceptions de messages et calculs internes) que les processeurs doivent exécuter pour résoudre une tâche distribuée. Enfin, nous appelons *spécification* un énoncé formel de la tâche à résoudre. En algorithmique distribué, les spécifications sont généralement formulées en deux propriétés : la *sûreté* et la *vivacité*. La sûreté d'une spécification correspond à l'ensemble des propriétés qui doivent être vérifiées en permanence durant une exécution d'un protocole. La vivacité d'une spécification correspond à l'ensemble des propriétés qui doivent être vérifiées à certains instants de l'exécution.

2.2 Les principales caractérisques d'un système distribué

Un système distribué diffère d'un système *centralisé* (*i.e.*, constitué d'une seule machine) par un certain nombre de caractériques. Ces caractéristiques montrent généralement certains avantages du

[1] Dans certains problèmes, les liens de communications peuvent être supposés unidirectionnels.

système distribué par rapport au système centralisé. Les principales caractéristiques d'un système distribué sont les suivantes (cette liste n'est pas exhaustive).

2.2.1 L'échange d'informations

L'une des motivations premières des systèmes distribués est de permettre les échanges de données notamment à grande échelle. L'exemple le plus marquant étant le réseau *Internet* qui permet à des milliards d'individus de communiquer et d'échanger de grandes masses de données. À l'origine, les réseaux avaient principalement des applications militaires : le but était de faire communiquer des centres de commandement géographiquement distants.

2.2.2 La distribution des données

Mettre en réseau plusieurs machines permet d'obtenir un espace disque conséquent pour un coût raisonnable. De plus, alors que l'espace disque d'une seule machine est généralement insuffisant pour stocker l'ensemble des applications et des données dont un utilisateur a besoin, avoir plusieurs disques sur le réseau permet d'avoir plus d'applications et de données disponibles pour les utilisateurs. Enfin, le fait de pouvoir distribuer sur un réseau des données a permis de faire évoluer les techniques de sauvegarde de données avec l'apparition de méthodes de *duplications de données* sur le réseau. Par exemple, les méthodes *RAID* permettent de reconstituer les informations perdues suite à la panne définitive d'une machine grâce aux duplicatas répartis sur le réseau.

2.2.3 Le partage des ressources

Intrinsectement, un réseau est un système multi-utilisateurs, multi-tâches. Partager les ressources (imprimantes, disques durs, etc.) est économiquement intéressant pour de tels systèmes : on limite le rapport entre le nombre de ressources et le nombre d'utilisateurs tout en essayant de conserver une qualité de service semblable à celle d'un système mono-utilisateur. Par exemple, avoir une imprimante disponible sur le réseau doit permettre à tout utilisateur d'imprimer des fichiers en un temps raisonnable. Cependant, le partage des ressources pose le problème de l'accès concurrent des utilisateurs ou des applications à celles-ci (cf. sous-section 2.3.5, page 16).

2.2.4 L'augmentation de la puissance de calcul

Les systèmes distribués permettent les calculs concurrents. Lorsque le temps d'exécution d'un calcul sur une machine mono-processeur est important, le temps d'exécution de ce même calcul sur un système distribué sera généralement réduit de manière signicative. Récemment, un certain nombre de réseaux dédiés au calcul intensif (calcul *out of core*) ont été développés comme par exemple les grilles de calcul. Une grille de calcul exploite la puissance de calcul (processeurs, mémoires, ...) de milliers d'ordinateurs interconnectés entre-eux en donnant l'illusion d'un ordinateur virtuel très puissant. Les grilles de calcul permettent de résoudre d'importants problèmes de calcul nécessitant des temps d'exécution très longs (calculs sur le repliement des protéines, le séquençage des génômes, etc.). Pour cela, les chercheurs ont développé des intergiciels (*middleware*), comme par exemple, Globus, CONDOR, ou DIET, (cf. [NMFS02]) permettant le dialogue entre les différents composants d'une grille tout en masquant la complexité des échanges inter-applications.

2.2.5 La résistance aux fautes

Une faute (ou *panne*) correspond à une défaillance temporaire ou définitive d'un composant du système (processeur ou canal). Contrairement à un système centralisé, dans un système distribué, une partie des services peut continuer à fonctionner suite à une panne. Cependant, la multiplicité des composants d'un système accroît le risque que l'un d'entre eux tombe en panne. Ces défaillances influent sur le comportement des protocoles. Donc, en cas de fautes, la validité des résultats calculés par un protocole ne peut pas toujours être assurée. Toutefois, nous verrons dans le chapitre 3 qu'il existe deux types de protocoles tolérants aux fautes : d'une part, les protocoles robustes (sous-section 3.1) et, d'autre part, les protocoles stabilisants (sous-section 3.2). Ces protocoles permettent d'assurer une certaine continuité de service malgré les pannes.

2.3 Quelques problèmes fondamentaux

L'exploitation des systèmes distribués ainsi que l'utilisation d'applications réparties posent des problèmes algorithmiques fondamentaux liés aux caractéristiques et aux contraintes de tels systèmes. Nous présentons maintenant une liste non exhaustive de ces problèmes fondamentaux.

2.3.1 La synchronisation

La synchronisation est l'action de coordonner plusieurs opérations entre elles en fonction du temps. Faire l'hypothèse qu'un système distribué est *synchrone* revient à supposer que :

1. Il existe une horloge globale à laquelle tous les processeurs ont accès.
2. Le temps de tranfert des messages dans les canaux est borné.

Cependant, les contraintes physiques des systèmes distribués ne garantissent pas toujours ces deux conditions. Dans ce cas, le système est dit *asynchrone*. Awerbuch a introduit la notion de *synchronisation* dans [Awe85a]. Ce type d'algorithme permet d'implanter des algorithmes synchrones dans un système asynchrone et assure que les actions de l'algorithme synchrone sont exécutées par *phases* (aussi appelées *pulsations*).

2.3.2 L'élection

De nombreux algorithmes répartis sont fondés sur l'hypothèse qu'il existe un processeur distingué, appelé *leader*, qui aura un comportement différent des autres dans le protocole. Or, l'un des objectifs principaux dans la réalisation d'algorithmes distribués est de pouvoir interchanger le rôle d'un ou plusieurs composants du système. Donc, pour certains problèmes, il peut être nécessaire de distinguer dynamiquement au cours d'une exécution un (nouveau) processeur parmi tous les processeurs du réseau : c'est le rôle d'un protocole d'élection. Il existe de nombreux protocoles d'élection dans la littérature parmi lesquels ceux de Le Lann [LL77] et de Chang et Roberts [CR79] pour des réseaux en anneau et celui de Gallager, Humblet et Spira pour des réseaux quelconques [GHS83].

2.3.3 Le calcul d'un état global

L'état global d'un système distribué dépend de la valeur des variables locales de chaque processeur ainsi que des messages contenus dans les liens de communications. Dans un système distribué asynchrone, aucun processeur ne peut obtenir un état global du système de manière instantanée. Le problème est donc de collecter des données globales pertinentes sur le système durant une période finie. Ces données ne correspondent pas forcément à un état réel du système à un instant donné

mais doivent permettre d'en vérifier la cohérence. Des protocoles de calcul d'état global (*snapshot protocols*) sont proposés dans [LY87, Mat88].

2.3.4 La détection de terminaison

Détecter la terminaison d'un algorithme distribué est un problème important mais non trivial. Dans de nombreux algorithmes distribués, un sous-ensemble restreint de processeurs peut décider de la terminaison d'un protocole sans utiliser de mécanisme extérieur : par exemple, dans les algorithmes de parcours (cf. section 5.2), l'initiateur du parcours détecte en un temps fini la terminaison du parcours. Cependant, il existe des algoritmes distribués pour lesquels il n'existe pas de détection de terminaison intrinsèque : par exemple, le protocole de Toueg [Tou80] pour le calcul des plus court chemins. Dans ce cas, il peut donc être nécessaire de mettre en place un protocole de contrôle qui observe le calcul sous-jacent et notifie la terminaison du calcul en un temps fini après qu'elle se soit produite à un ou plusieurs processeurs. Cependant, l'inconvénient majeur lorsqu'on utilise un protocole externe de détection de terminaison est que son surcoût en temps est non bornable par rapport à la complexité du calcul sous-jacent à observer. De nombreux travaux ont été effectués sur la détection de terminaison, par exemple [Fra80, DS80, Mat87].

2.3.5 L'allocation de ressources

Un des intérêts principaux des réseaux est le partage des ressources (cf. section 2.1). Or, ces ressources sont en nombre limité. Donc, permettre aux processeurs un accès exclusif et équitable aux ressources qu'ils demandent est un problème crucial des systèmes distribués. L'un des problèmes de base d'allocations de ressources est l'*exclusion mutuelle* [Dij65] : il s'agit de partager une unique ressource commune à tous les utilisateurs du réseau. Ce problème à déjà été traité dans le cadre des systèmes centralisés multi-programmés (l'ouvrage de Beauquier et Bérard présente les concepts fondamentaux de tels systèmes [BB90]). Les protocoles d'exclusion mutuelle sont classés en deux catégories : ceux fondés sur des demandes de permission d'entrer en section critique [Lam78, RA81] et ceux fondés sur la circulation perpétuelle d'un jeton [LL77, NT87].

2.3.6 L'interblocage

Le problème d'interblocage apparaît essentiellement dans le cadre de l'allocation de ressources. L'interblocage (*deadlock*) se produit lorsque des processeurs s'attendent mutuellement [Ray85]. Cela peut être du à :

- Un circuit dans l'attente de message. Par exemple, un processeur p_1 attend un message de p_2 qui attend un message de p_3 et p_3 attend un message de p_1.
- Un problème de partages de ressources. Par exemple, p_1 détient la ressource A et souhaite obtenir la ressource B tandis que p_2 détient B et souhaite obtenir A. Problème : p_1 libèrera A lorsqu'il obtiendra B et, réciproquement, p_2 libèrera B lorsqu'il obtiendra A.

Les processus bloqués dans cet état le sont définitivement. Un interblocage est traité soit *a priori* en évitant son apparition (méthode pessimiste) soit a posteriori en le détectant et en l'éliminant (méthode optimiste). Les principaux travaux relatifs au problème d'interblocage peuvent être trouvés dans [CMH83, MM79, MM88].

2.3.7 Le routage

Le routage est le mécanisme permettant d'acheminer des données d'un processeur à un autre, même s'ils ne sont pas voisins. Classiquement, deux méthodes sont utilisées pour résoudre ce pro-

blème : la *diffusion* (ou *inondation*) [CM82] et la gestion de *tables de routages* (aussi appelés *annuaires*) [Taj77, Tou80].

Chapitre 3

La tolérance aux fautes

L'un des enjeux fondamentaux de l'algorithmique distribuée est la résistance aux fautes (cf. section 2.2). Une faute (ou *panne*) désigne une défaillance définitive ou temporaire d'un ou plusieurs composants du système (processeurs ou liens de communications). Les fautes sont généralement classées suivant certains critères [Nol02] :

- *L'origine de la faute*, c'est à dire, le type de composant qui est responsable de la faute, lien de communication ou processeur.
- *La cause de la faute*. Pour ce critère, on distingue les fautes *par omission* et les fautes *byzantines*. Les fautes par omission regroupent les défaillances des composants alors que les fautes byzantines sont dûes à des processeurs *byzantins* qui ont un comportement arbitraire ne suivant plus le code de leurs algorithmes locaux.
- *La durée de la faute*. Si la durée d'une faute est supérieure au temps d'exécution du protocole, elle est dite *définitive* Faute sinon elle est dite *transitoire* ou *intermittente*.
- *La détectabilité de la faute*. Une faute est *détectable* si son incidence sur la cohérence de l'état d'un processeur permet à celui-ci de s'en apercevoir.

De nombreux algorithmes tolérants aux fautes sont proposés dans la littérature. Ils peuvent être classés en deux catégories [Tel01] :

- Les algorithmes *robustes* (section 3.1) qui abordent le problème de tolérance aux fautes selon une approche *pessimiste* où les processeurs suspectent toutes les informations qu'ils recoivent.
- Les algorithmes *stabilisants* (section 3.2) qui abordent le problème selon une approche *optimiste*. Cette méthode peut causer un comportement anormal des processeurs non-défaillants mais garantit le retour vers un comportement global normal en un temps fini après que les fautes ont cessé.

3.1 Les algorithmes robustes

Un algorithme est dit *robuste* lorsqu'il garantit que les processeurs non défaillants continuent à se comporter correctement malgré les fautes se produisant sur d'autres processeurs pendant l'exécution [Tel01]. De tels algorithmes doivent toujours vérifier la spécification du problème à résoudre. Par exemple, un algorithme robuste d'élection garantit, en dépit des fautes, qu'un unique processeur non défaillant sera élu en un temps fini.

Moran et Wolfstahl ont montré dans [MW87] que dans un environnement tolérant aux fautes, la plupart des problèmes évoqués dans la section 2.3, comme par exemple, l'élection de leader, peuvent se réduire au *problème du consensus* (ou *prise de décision commune*). Dans le problème du *consensus*, chaque processeur a une donnée initiale binaire et doit calculer en un temps fini un résultat lui aussi de type binaire. Les résultats obtenus sur chaque processeur doivent être identiques (les proces-

seurs doivent se mettre en accord) même si les données initiales sont différentes. Or, Fisher, Lynch et Paterson ont montré en 1985 dans [FLP85] que le problème du consensus n'admet pas de solution déterministe dans un système distribué asynchrone si une faute définitive se produit sur un processeur. Cette preuve d'impossibilité est fondée sur le fait que dans un système asynchrone, les voisins d'un processeur lent ne peuvent pas décider si ce processeur est effectivement lent ou s'il est en panne. Ce résultat important induit que sans hypothèses supplémentaires sur le modèle (par exemple, supposer que le temps d'acheminement des messages est borné), sur les fautes (par exemple, supposer que les fautes sont détectables) ou sur les spécifications (authoriser une perte de sûreté temporaire), le problème du consensus ne peut pas être résolu de manière déterministe dans un système asynchrone dans lequel des fautes peuvent survenir.

3.2 Les algorithmes stabilisants

La notion de *stabilisation* regroupe deux paradigmes : *l'auto-stabilisation* définie par Dijkstra en 1974 [Dij74] et la *stabilisation instantanée* définie par Bui, Datta, Petit et Villain en 1999 [BDPV99c].

3.2.1 L'auto-stabilisation

Dijkstra énonce dans [Dij74] qu'un système est auto-stabilisant si et seulement si, quel que soit son état initial, il est capable de retrouver, de lui même, un état *légitime* en un nombre fini d'étapes. Nous appelons état légitime, un état à partir duquel le système fonctionne correctement, *i.e.*, un état à partir duquel la spécification du problème à résoudre est vérifiée. Réciproquement, le système est dans un état *illégitime* lorsque la spécification du problème à résoudre n'est pas vérifiée à partir de cet état.

À partir d'un état initial quelconque (c'est à dire un état où les variables ne sont pas initialisées et ont donc des valeurs arbitraires), un algorithme auto-stabilisant vérifie deux propriétés :

- La *convergence*. La propriété de convergence assure qu'à partir de n'importe quel état initial, le système atteint en un temps fini un état légitime. La phase de convergence de l'algorithme est aussi appelée phase de *stabilisation*.
- La *clôture*. La propriété de clôture assure que tout état atteint à partir d'un état légitime est lui aussi légitime.

Ces deux propriétés impliquent que les algorithmes auto-stabilisants sont tout à fait adaptés pour tolérer les fautes transitoires à condition qu'elles ne modifient pas le code de l'algorithme. En effet, après une perturbation transitoire de la mémoire ou des liens de communications, ces composants peuvent contenir des informations illégitimes vis-à-vis des spécifications du problème à résoudre (cf. Figure 3.2.1). Un algorithme auto-stabilisant garantit qu'en un temps fini si aucune autre faute ne vient perturber le système[1] alors le système convergera de lui-même vers un état légitime vis-à-vis du problème à résoudre. Nous appelons *temps de stabilisation* d'un algorithme, le temps de convergence maximal du système à partir d'un état initial quelconque.

Avantages et inconvénients des algorithmes auto-stabilisants. Les algorithmes auto-stabilisants offrent un certain nombre d'avantages vis-à-vis des algorithmes distribués classiques [Tel01] :

- *La tolérance aux fautes*. Comme nous l'avons vu précédemment, l'auto-stabilisation est une technique générale permettant d'écrire des protocoles tolérant les fautes transitoires. En effet, suite à une faute transitoire, l'état d'un système peut être quelconque. À partir de cet état, un

[1] Les fautes transitoires sont supposées rares.

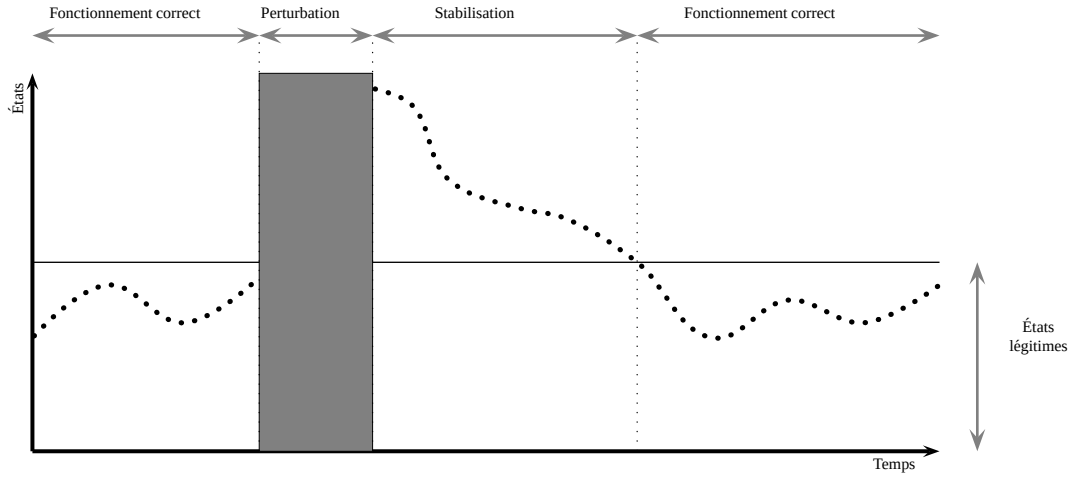


FIG. 3.2.1 – Système auto-stabilisant

protocole auto-stabilisant convergera vers un état légitime du système sous réserve qu'aucune nouvelle faute ne vienne à nouveau le perturber. En conséquence, pour étudier la convergence, l'état initial quelconque du système est supposé correspondre à l'état immédiatement postérieur à la fin de la dernière faute transitoire et à partir de cet état, nous considérons que plus aucune faute n'apparaîtra dans le système. Cette hypothèse peut sembler forte mais l'impossibilité en général de détecter les fautes dans les systèmes asynchrones ([FLP85]) nous oblige à faire cette hypothèse. De plus, les fautes transitoires sont supposées rares et durent un temps fini, donc, cela signifie qu'entre deux fautes le système a le temps de converger et d'à nouveau fournir des services cohérents.

- *L'absence d'initialisation*. Dans l'algorithmique distribuée classique, l'initialisation est une phase critique. Or, cette phase n'existe pas en auto-stabilisation car un système auto-stabilisant converge quel que soit son état initial.
- *La résistance aux changements topologiques*. De nombreux algorithmes auto-stabilisants, notamment les protocoles de calcul de tables de routages et de calcul d'arbres couvrants, s'adaptent sous certaines conditions aux *systèmes dynamiques*, c'est à dire, aux systèmes dont la topologie peut changer pendant l'exécution de l'algorithme.

Cependant, l'auto-stabilisation a quelques inconvénients :

- *La perte temporaire de sûreté*. Durant, la phase de stabilisation, le système ne garantit pas que les résultats calculés par le protocole sont corrects.
- *Un surcoût en temps et en espace*. Le surcoût en temps correspond au temps d'exécution une fois le protocole stabilisé. En effet, le temps d'exécution d'un protocole auto-stabilisant est généralement plus élevé en marche "normale" qu'un protocole non auto-stabilisant. Ensuite, les techniques utilisées pour obtenir des solutions auto-stabilisantes ont généralement besoin de mémoire supplémentaire.
- *L'absence de détection de stabilisation*. L'auto-stabilisation est une propriété sur le système dans son ensemble. Donc, en général, il n'est pas possible d'observer localement si le système est dans un état légitime ou illégitime.

Les classes d'algorithmes auto-stabilisants. Suite à l'article fondateur de Dijkstra [Dij74], de nombreux protocoles auto-stabilisants ont été proposés pour résoudre une grande variété de pro-

blèmes, par exemple, [Kru79] pour la propagation d'informations avec retour (PIR), [HC93, Var93] pour la circulation de jeton en profondeur, [DIM93] pour l'exclusion mutuelle et [CD94, AB98] pour le calcul d'arbre couvrant. Classiquement, la plupart des protocoles auto-stabilisants peuvent être classés en deux grandes familles de protocoles :

– *Les protocoles à vagues*. Un protocole à vagues vérifie les trois propriétés suivantes :

1. Tous les cycles de calcul, appelés *vagues*, du protocole se déroulent en temps fini.
2. Pour chaque cycle de calcul du protocole, il existe au moins un processeur qui exécute une action particulière dite de *décision*.
3. Pour chaque cycle de calcul du protocole, les actions de *décision* sont causalement précédées par au moins une action sur chaque processeur.

Pour assurer la propriété de convergence, les algorithmes à vagues auto-stabilisants exécutent généralement une infinité de vagues, dans ce cas, ils sont appelés *algorithmes à vagues perpétuelles* ou encore *algorithmes à vagues cycliques*. De nombreux protocoles à vagues auto-stabilisants sont proposés dans la littérature, par exemple, pour le problème de la circulation de jeton en profondeur (cf. [HC93, JB95, JABD97, DJPV00]) et pour la propagation d'information avec retour (PIR) (cf. [Kru79, DIM97, BDPV99b, Var93, CDPV01b]).

– *Les protocoles silencieux*[2]. Un protocole est dit silencieux s'il converge vers un état global silencieux où les valeurs des variables locales de processeurs ne changent plus [DGS96]. Les protocoles silencieux auto-stabilisants sont utilisés pour résoudre des problèmes de calcul de point fixe comme, par exemple, le calcul d'arbre couvrant (cf. [CYH91, Her91a, CD94, AB98, DT01]) et le calcul de propriétés sur le réseau comme les points d'articulation ou les isthmes (cf. [Kar99, KC99, Cha99b, Cha99a, Dev05]).

3.2.2 La stabilisation instantanée

Dans les systèmes distribués, les *actions de démarrage* correspondent aux actions particulières qui initient de nouveaux cycles de calcul d'un protocole dans le réseau. Par exemple, pour le protocole classique (non-tolérant aux fautes) de circulation de jeton [Cha82], l'action de démarrage correspond à l'action de l'initiateur qui crée un jeton et l'envoie à un de ses voisins.

Dans les protocoles non tolérants aux fautes, la notion d'action de démarrage est liée à la notion d'état initial du système. Implicitement, lorsque le protocole démarre, on suppose que :

1. Le système est dans un état initial pré-défini.
2. Une action de démarrage est exécutée lorsqu'une requête extérieure au protocole (cette requête est émise par une application ayant besoin du service fournit par le protocole, cf. Figure 3.2.2) a demandé un cycle de calcul.

Les actions de démarrage d'un protocole non tolérants aux fautes sont essentielles du fait qu'elles sont à la base de l'analyse du protocole même si cela est souvent implicite. En effet, le déroulement du protocole est observé à partir de l'exécution de l'une de ces actions.

Dans les protocoles auto-stabilisants, comme l'état initial du système est quelconque, les premières actions exécutées sont elles aussi quelconques : en conséquence n'importe quelle action est susceptible de s'exécuter lors du premier pas de calcul. De plus, en auto-stabilisation, on s'intéresse à la convergence du système vers un comportement spécifique en un temps fini. Assurer cette convergence nécessite la répétition des cycles de calcul à l'infini. Donc, puisque les cycles de calcul sont répétés à l'infini, les notions de requête et d'action de démarrage disparaissent purement et simplement dans l'analyse de tels systèmes. Seuls l'état initial, toujours point de départ de l'étude, et

[2]Contrairement à la classe des protocoles à vagues qui a été définie pour les algorithmes distribués en général, la classe des protocoles silencieux a été introduite spécialement pour les systèmes auto-stabilisants.

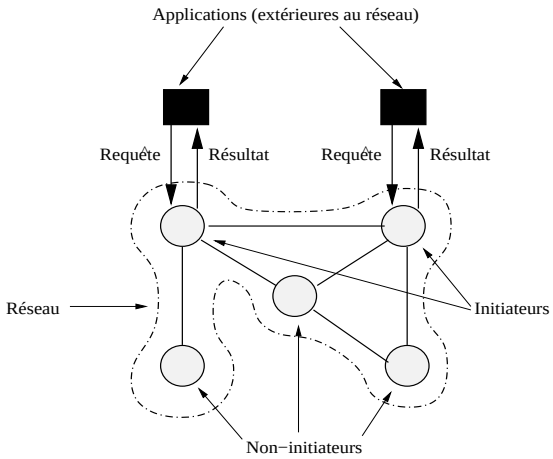


FIG. 3.2.2 – Niveau “application” dans un réseau

l’ensemble des états légitimes, buts de la convergence, sont alors pris en compte dans l’analyse des protocoles auto-stabilisants.

Au contraire, dans les protocoles *instantanément stabilisants*, les actions de démarrage sont au moins aussi importantes que l’état initial du système. Le concept de *stabilisation instantanée* [BDPV99c] a été défini en 1999 par Bui, Datta, Petit et Villain comme suit :

Définition 3.2.1 (Stabilisation instantanée) *Quel que soit l’état initial du système, un protocole* instantanément stabilisant *garantit que le système satisfait toujours ses spécifications.*

Plus précisément, un protocole instantanément stabilisant garantit qu’à partir de n’importe quel état initial du système, lorsque le système a besoin qu’un protocole réalise une tâche :

1. le protocole est initié par au moins une action de démarrage en un temps fini.
2. Suite à une action de démarrage, le calcul initié est exécuté conformément à la spécification de la tâche.

Donc, pour analyser de tels protocoles, les actions de démarrage doivent être obligatoirement explicitées.

En auto-stabilisation, le temps de stabilisation est considéré comme étant le temps maximal, partant d’un état initial quelconque, avant que le système vérifie ses spécifications. Le temps de stabilisation peut donc être décomposé en : le temps partant de l’état initial avant la première action de démarrage (aussi appelé *délai*, cf. section 3.2.3) plus le temps d’exécution des cycles de calcul non conforme aux spécifications. En stabilisation instantanée, un autre point de vue est proposé : tant que le système ne démarre pas un calcul effectivement demandé (via une action de démarrage), le système vérifie ses spécifications. Par exemple, un protocole de PIR peut être spécifié comme suit : lorsqu’un initiateur a un message m à envoyer, la diffusion de m commence en un temps fini, ensuite, tout processeur recoit m en un temps fini et l’initiateur recevra en un temps fini des accusés de reception de m de la part de tous les autres processeurs. Cette spécification est trivialement vérifiée tant qu’aucun initiateur n’a de message à diffuser. Donc, cela signifie que le délai n’est pas à prendre en compte dans le temps de stabilisation et comme un protocole instantanément stabilisant assure que dès la première action de démarrage, les calculs effectués par le protocole sont conformes aux spécifications du calcul demandé, le temps de stabilisation est nul. D’où, tout protocole instantanément stabilisant est optimal en temps de stabilisation. Plus simplement, cela signifie que, contrairement à l’auto-stabilisation où le résultat attendu n’est assuré qu’après avoir répété un nombre fini de fois des cycles

de calcul du protocole[3], un protocole instantanément stabilisant assure que le calcul demandé est effectué conformément aux spécifications dès la première action de démarrage.

Bien sûr, dans un système instantanément stabilisant, l'état initial ainsi qu'un certain nombre des états suivants peuvent ne pas correspondre aux états générés par le système s'il avait été "proprement" initialisé. Cependant, contrairement aux systèmes auto-stabilisants, ces états "non souhaités" n'empêchent pas le système de vérifier ses spécifications. L'approche instantanément stabilisante permet ainsi de limiter l'absence de sûreté à la période (*a priori* incontrôlable) correspondant à l'existence de fautes transitoires dans le système.

Spécifications. En 2002, lors des journées internationales sur l'auto-stabilisation [Vil02], Villain a montré que le concept de stabilisation instantanée est fortement lié à la manière dont le problème à résoudre est spécifié. Généralement, les systèmes auto-stabilisants sont spécifiés de façon "statique" : un sous-ensemble d'états légitimes est défini parmi les états possibles du système correspondant aux états où une propriété voulue est sûre d'être vérifiée. Ces systèmes convergent vers un état légitime quel que soit l'état initial du système. Par exemple, la sûreté du problème de l'exclusion mutuelle est définie comme suit : "il n'y a jamais plus d'un processeur en section critique". Donc, l'ensemble des états vérifiant la sûreté correspond à un sous-ensemble des états où au plus un processeur est en section critique. En utilisant de telles spécifications, il est *a priori* impossible de trouver des solutions instantanément stabilisantes (sauf si l'ensemble des états légitimes est égal à l'ensemble des états du système comme dans [JADT02]). En effet, si l'état initial du système n'est pas un état légitime, la spécification du problème n'est pas vérifiée, et, par définition, le système n'est pas instantanément stabilisant. En fait, un système instantanément stabilisant est généralement spécifié de manière "dynamique" : il n'y a pas de distinction parmi les états du systèmes, on s'intéresse plutôt au déroulement de l'exécution du système au cours du temps. Le système est instantanément stabilisant si et seulement si, durant l'exécution, à chaque requête, la tâche correspondante est initiée (via une action de démarrage) en un temps fini dans le système et, suite à cette amorce, les actions exécutées dans le système effectuent la tâche demandée. Par exemple, un protocole d'exclusion mutuelle instantanément stabilisant assure que si un processeur devient demandeur de la section critique au cours de l'exécution, celui-ci l'obtiendra, par la suite, en un temps fini (*vivacité*) et lorsqu'il entrera en section critique, aucun autre processeur ne sera ou ne pourra entrer en section critique (*sûreté*). Dans [CDPV03], Cournier, Datta, Petit et Villain montrent que tout problème ayant une solution auto-stabilisante dans un réseau non uniforme peut être spécifié de manière dynamique et peut être résolu de manière instantanément stabilisante.

La stabilisation instantanée apporte des éclairages différents en fonction du type de problème à résoudre. Les problèmes classiques de l'algorithmique distribuée peuvent être classés en deux ensembles de problèmes : les problèmes de *service* et les problèmes de *calcul de point fixe*.

Service. Un problème de service consiste à fournir en un temps fini un *service* aux processeurs qui le demandent. Un service est une tâche spécifique dont l'exécution dépend du processeur demandeur. Ce type de problème peut être spécifié comme suit :

1. À partir de n'importe quel état du système, si un processeur demande le service alors ce service est initié en un temps fini via une action de démarrage.
2. Suite à une action de démarrage d'un processeur p, le service initié est exécuté vis-à-vis de p conformément à sa spécification.

Par exemple, l'exclusion mutuelle est un problème de service : durant l'exécution d'un protocole d'exclusion mutuelle, des processeurs peuvent demander l'accès à la section critique. En conséquence, ils doivent l'obtenir en un temps fini et de manière exclusive.

[3]Ce nombre est en général inconnu et non borné.

Remarque 3.2.1 *Soient $\mathcal{A}$ un service et $\mathcal{SP}_{\mathcal{A}}$ une spécification de $\mathcal{A}$. Pour prouver qu'un protocole de service $\mathcal{P}$ (*i.e. *un protocole défini pour résoudre un problème de service) est instantanément stabilisant pour la spécification $\mathcal{SP}_{\mathcal{A}}$, nous devons montrer que toute exécution de $\mathcal{P}$ vérifie les deux conditions suivantes :*

1. *Tout service $\mathcal{A}$ demandé est initié en un temps fini.*
2. *À partir de n'importe quel état du système où $\mathcal{A}$ est initié, le calcul de $\mathcal{A}$ (par $\mathcal{P}$) est conforme à la spécification $\mathcal{SP}_{\mathcal{A}}$.*

Pour les protocoles de service, l'avantage prépondérant de la stabilisation instantanée vis à vis de l'auto-stabilisation est le suivant. En auto-stabilisation, le cycle de calcul initié par la première requête n'est pas forcément conforme à la spécification du service demandé. Donc, si l'application (ou l'utilisateur) désire qu'au moins un service finisse par être exécuté conformément à ses spécifications, les requêtes doivent être répétées. Or, dans le cas général, les solutions auto-stabilisantes assurent que le service demandé ne sera exécuté conformément à ses spécifications qu'après un nombre fini mais inconnu (voire non borné) de requêtes. En conséquence, les requêtes doivent être répétées à l'infini et le protocole devient cyclique (c'est pourquoi la notion de requêtes est implicite en auto-stabilisation). Au contraire, une solution instantanément stabilisante pour le même problème assurera que le service sera rendu conformément à ses spécifications dès la première requête et, donc, l'application n'aura pas besoin de répéter les requêtes. En particulier et contrairement aux protocoles auto-stabilisants, tout protocole de service instantanément stabilisant détecte la terminaison d'un service : il détecte lorsqu'un service a été rendu conformément à ses spécifications.

Depuis l'article fondateur de Bui, Datta, Petit et Villain ([BDPV99c]), de nombreux protocoles instantanément stabilisants ont été proposés pour résoudre des problèmes de service : par exemple, pour le problème de la circulation de jeton en profondeur (cf. [PV03, CDPV06, CDV05b]) pour la propagation d'information avec retour (PIR) (cf. [CDPV02, BCV03, CDV06b]) et la synchronisation de phase (cf. [Nol02]).

Calcul de point fixe. Un problème de point fixe peut être codé symboliquement par une variable $\mathcal{X}$ sur chacun des processeurs d'un sous-ensemble donné. $\mathcal{X}$ peut être simple (par exemple, pour le calcul d'un arbre couvrant enraciné : un "pointeur père" sur chacun des processeurs sauf la racine) ou complexe (par exemple, dans le calcul d'une table de routage : l'ensemble des couples "numéro de canal/identité" de chacun des processeurs). Un algorithme calculant un point fixe doit converger en un temps fini vers une valeur constante de la variable $\mathcal{X}$ pour chacun des processeurs du sous-ensemble, ces valeurs devant vérifier la propriété qui définit le point fixe à calculer. Nous allons maintenant définir formellement les spécifications d'un tel problème. Soit $\mathcal{F}$ un problème de point fixe défini sur les variables $\mathcal{X}$ d'un sous-ensemble V' de processeurs du système. Tout protocole $\mathcal{P}$ calculant $\mathcal{F}$ vérifie :

- À partir de son état initial, toute exécution e de $\mathcal{P}$ converge en un temps fini vers un suffixe e' dans lequel la valeur de $\mathcal{X}$ est constante sur tous les processeurs de V' (*Vivacité*).
- Dans tous les états de e', les variables $\mathcal{X}$ des processeurs de V' vérifient la propriété définissant $\mathcal{F}$ (*Sûreté*).

Par exemple, l'élection de leader peut être ramenée à une variable booléenne "Élu" sur chacun des processeurs : dès lors, la propriété que doivent vérifier les variables $\mathcal{X}$ dans e' est qu'il y a une et une seule valeur VRAI dans le système.

Il faut noter que la plupart des problèmes de point fixe peuvent être résolus par des protocoles de service. En effet, la projection des variables d'un protocole de service restreinte aux variables implantant le problème de point fixe sous-jacent converge vers un suffixe stable (*i.e.*, où les valeurs de variables ne changent plus) où la propriété définissant le point fixe est vérifiée. Par exemple, un

protocole de circulation de jeton en profondeur peut être utilisé pour calculer un arbre couvrant en profondeur du réseau. En général, une fois qu'il est stabilisé, la circulation réalisée par le protocole devient unique. Or comme les variables implémentant l'arbre couvrant (généralement des pointeurs "pères") sont calculées dynamiquement en fonction de l'ordre des visites de la circulation, les valeurs de ces variables finissent par ne plus jamais être modifiées.

Nous allons maintenant montrer que tout protocole auto-stabilisant de calcul de point fixe est aussi instantanément stabilisant. Considérons un protocole auto-stabilisant $\mathcal{P}$ pour un calcul de point fixe. Par définition, à partir de n'importe quel état initial, toute exécution de $\mathcal{P}$ converge vers un suffixe où la valeur de la variable $\mathcal{X}$ (simple ou composite) de chaque processeur de V' est constante. Donc, la vivacité de sa spécification est toujours vérifiée au cours de l'exécution. Ensuite, nous pouvons remarquer que la sûreté est de la forme $A \Rightarrow B$ où A correspond à "l'exécution a atteint le suffixe" et B correspond à "les variables $\mathcal{X}$ des processeurs de V' vérifient la propriété définissant le point fixe". Tant que l'exécution n'a pas atteint le suffixe (*i.e.*, tant que l'exécution converge), la prémisse A de l'implication $A \Rightarrow B$ est fausse, donc, cette implication est satisfaite. Enfin, lorsque l'exécution atteint le suffixe, A et B sont vrais et donc l'implication $A \Rightarrow B$ est elle aussi vraie. Ainsi, quel que soit l'état initial du système, la sûreté est toujours vérifiée au cours d'une exécution de $\mathcal{P}$. D'où, tout protocole auto-stabilisant pour un calcul de point fixe satisfait toujours sa spécification quel que soit l'état initial du système et, par définition, un tel protcole est instantanément stabilisant. En particulier, par définition, les protocoles auto-stabilisants silencieux résolvent des problèmes de point fixe. Donc, tous les protocoles auto-stabilisants silencieux, comme par exemple [CYH91, CD94, AB98, Kar99, Dev05], sont instantanément stabilisants. Ce résultat, qui semble notamment en contradiction avec l'article de Aggarwal et Kutten [AK93], reflète en fait un problème de vocabulaire où les auteurs considèrent comme états légitimes uniquement les états vérifiant la propriété recherchée. Ce point de vue implique, en particulier, que tout algorithme classique de calcul de point fixe conçu pour un environnement sans faute ne vérifie jamais la sûreté en cours de calcul mais seulement dans l'état terminal. Il y aurait donc confusion entre la notion de temps de calcul (temps nécessaire pour obtenir un état terminal) et celle de temps de convergence (temps nécessaire pour obtenir un état légitime) (cf. [Vil02]). Cependant, bien que ces protocoles stabilisent en temps zéro, le temps de calcul des protocoles silencieux peut être, dans certain cas, problématique. En effet, si quelques fautes transitoires viennent perturber le système après qu'il ait atteint un état terminal, cela peut provoquer un recalcul quasi-complet pour atteindre à nouveau un état terminal. La notion de *fault containment* a été introduite afin de palier à ce problème : le *fault containment* [GGHP00] est la faculté de certains protocoles auto-stabilisants à limiter les effets des fautes lorsqu'elles sont peu nombreuses dans le réseau.

État de l'art sur la stabilisation instantanée. Le concept de *stabilisation instantanée* [BDPV99c] a été introduit en 1999 par Bui, Datta, Petit et Villain avec un protocole de propagation d'information avec retour (PIR) pour des réseaux en arbres. Suite à cet article fondateur, de nombreux protocoles instantanément stabilisants ont été proposés, en particulier des protocoles à vagues. Il faut noter que, jusqu'à présent, toutes les solutions proposées sont écrites dans le modèle à états (cf. chapitre 4, page 31) : un modèle à mémoire partagée où l'envoi de messages est simulé par des accès directs en lecture aux variables des voisins.

Deux protocoles de parcours en profondeur instantanément stabilisants pour des réseaux en arbres sont présentés dans [PV99]. Le premier fonctionne dans un arbre orienté. Son délai est en $2h$ rondes (cf. sous-section 3.2.3 pour la définition du délai et définition 4.3.1, page 35, pour la définition des rondes) dans le pire des cas où h est la hauteur de l'arbre. Il nécessite $\Delta_p + 1$ états pour les noeuds internes et 2 états pour les feuilles. Le second fonctionne dans un arbre non-orienté. Son délai est de $3h$ rondes dans le pire des cas. Il nécessite $\Delta_p + 2$ états pour les noeuds internes (seulement $\Delta_p + 1$

pour la racine) et 2 états pour les feuilles. Ces deux protocoles sont optimaux en nombre d'états.

Le premier protocole de PIR instantanément stabilisant a été présenté dans [BDPV99c] pour des arbres orientés. Ce protocole est aussi optimal en nombre d'états : 3 états pour les noeuds internes, 2 pour la racine et les feuilles. Dans le pire des cas, son délai est en $2h$ rondes.

Un protocole instantanément stabilisant de PIR pour les arbres non-orientés est proposé dans [BDPV99a]. Dans le pire des cas, son délai est en $3h$ rondes. Ce protocole nécessite $\Delta_p + 2$ états par noeud interne p ainsi que 2 états pour la racine et les feuilles. Il n'est pas optimal en mémoire. Dans [CDPV01a], les auteurs montrent qu'il est impossible d'obtenir un protocole de PIR instantanément stabilisant pour arbre non-orienté avec seulement 3 états par noeud interne. Le nombre minimum d'états nécessaire pour obtenir une solution instantanément stabilisante pour le PIR dans l'arbre non-orienté est 4 pour les noeuds internes. Un tel protocole est aussi fourni dans [CDPV01a].

Le premier protocole de PIR instantanément stabilisant pour des réseaux quelconques est présenté dans [CDPV02]. L'inconvénient de cette solution est que le protocole nécessite la connaissance de la taille exacte du réseau. Donc, cette taille doit être constante. De ce fait, le protocole ne fonctionne pas dans les réseaux dynamiques. Cet inconvénient est résolu dans [BCV03] en utilisant une composition de trois algorithmes.

Dans [CDPV03], Cournier *et al* ont proposé des solutions instantanément stabilisantes pour quatre problèmes fondamentaux : *réinitialisation du réseau*, *calcul d'état global du système*, *élection de leader* et *détection de terminaison*. Ces solutions sont basées sur un protocole instantanément stabilisant de PIR pour réseaux quelconques. En utilisant ces protocoles fondamentaux, les auteurs montrent ensuite comment réaliser un protocole générique capable de fournir une version instantanément stabilisante d'une large classe de protocoles.

Enfin, il faut noter que des solutions instantanément stabilisantes ont été proposées sur la synchronisation de phase ([Nol02]), la diffusion simple [JADT02] dans un arbre et la construction d'arbres binaires de recherche ([BDV05]).

3.2.3 Étude des performances

La *complexité* est l'étude formelle des performances d'un algorithme. La complexité s'intéresse aux performances des algorithmes en termes de *temps* et d'*espace*. Deux critères sont importants dans d'évaluation d'une complexité d'un protocole :

1. La dépendance de cette complexité vis-à-vis de caractéristiques du réseau comme, par exemple, le nombre de sommets, le degré des processeurs, le diamètre, etc. En fonction de ces dépendances, nous pouvons établir des comparaisons entre les protocoles. Un protocole dont la complexité dépend du nombre de processeurs sera généralement plus coûteux qu'un protocole dont la complexité dépend du degré des processeurs (*i.e.*, le nombre de voisins des processeurs). De plus, le fait qu'une complexité dépende de paramètres locaux (par exemple, le degré des processeurs) ou de paramètres globaux (par exemple, le nombre de processeurs du réseau) a une incidence sur la dynamicité du réseau : un réseau dont la mémoire locale des processeurs dépend de paramètres locaux supportera plus facilement des modifications (par exemple, l'ajout de processeurs) qu'un réseau dont les besoins en mémoire locale des processeurs dépendent de caractéristiques globales du réseau.
2. Ensuite, les types des fonctions qui apparaissent dans cette complexité : constante, logarithmique, polynomiale, exponentielle. Ces fonctions sont très importantes pour établir l'ordre de grandeur de la complexité.

Complexité en temps. Dans cette thèse, nous restreignons notre étude aux protocoles de service. Historiquement, les chercheurs en auto-stabilisation se sont intéressés presque exclusivement au

temps de stabilisation pour comparer la complexité en temps des protocoles, notamment pour les protocoles de service. Par définition, les protocoles instantanément stabilisants ont un temps de stabilisation nul. Cependant, si nous observons les protocoles de service instantanément stabilisants, nous pouvons remarquer qu'un temps de stabilisation nul ne signifie pas que le service demandé sera initié imédiatement. Des critères plus pertinents sont donc nécessaires pour permettre une comparaison entre de tels protocoles. Le *délai* (qui est aussi évaluable dans les protocoles auto-stabilisants) a été introduit dans [BDPV99c] en même temps que la stabilisation instantanée et permet une telle comparaison. D'autres critères permettent d'analyser plus finement les protocoles de service stabilisants [Cou02]. La multiplicité de ces critères de comparaison peut permettre d'écrire des protocoles plus performants. Nous allons maintenant lister (cette liste n'est pas exhaustive) les critères que nous jugeons pertinents pour l'étude de la complexité en temps des protocoles stabilisants (auto-stabilisation ou stabilisation instantanée).

- **Le temps de stabilisation**. Partant de n'importe quel état initial du système, le temps de stabilisation d'un protocole correspond au temps maximal pour atteindre un état à partir duquel le système satisfait ses spécifications.
- **Le délai**. Partant de n'importe quel état initial du système, le *délai* correspond au temps maximal avant que la première action de démarrage soit exécutée. Cette mesure représente donc le temps nécessaire pour que le protocole initie le premier cycle de calcul. Par définition, seuls les protocoles instantanément stabilisants assurent que dès la première action de démarrage, le cycle de calcul initié soit effectué conformément à sa spécification. Par conséquent, pour un protocole instantanément stabilisant, cette mesure représente le temps nécessaire pour initier un cycle de calcul qui sera conforme aux spécifications.
- **Le nombre de fois où le protocole doit être initié pour obtenir un cycle de calcul conforme aux spécifications**. Par définition, pour n'importe quel protocole de service instantanément stabilisant, ce nombre est égal à un. Par exemple, un protocole de PIR instantanément stabilisant assure que lorsqu'un processeur p initie la diffusion d'un message :

  1. Ce message sera reçu en un temps fini par tous les autres processeurs du réseau.
  2. p recevra en un temps fini un accusé de réception de ce message venant de chacun des autres processeurs.

  Pour la plupart des protocoles auto-stabilisants, comme par exemple [HC93, CDPV01b, Pet01], le nombre de fois que le protocole doit être initié pour obtenir un cycle de calcul conforme aux spécifications peut ne pas être borné. En effet, les cycles de calcul initiés par ces protocoles peuvent s'effectuer de manière incomplète un nombre non borné de fois. Cela est dû au fait qu'il est possible d'avoir des exécutions de ces protocoles où certains comportements anormaux s'effacent très lentement alors que les cycles de calcul initiés s'effectuent rapidement en ignorant les comportement anormaux. Trouver des solutions auto-stabilisantes où le nombre d'initialisations nécessaires pour obtenir un cycle de calcul conforme aux spécifications est borné peut être très intéressant car cela permet de retrouver la sûreté en évitant d'itérer les cycles de calcul éternellement. Par exemple, si pour un protocole de PIR auto-stabilisant cette borne est égale à k, alors l'initiateur peut assurer que tous les autres processeurs recevront un message m au moins une fois en initiant la diffusion de m au moins k fois et, par extension, on obtient un protocole instantanément stabilisant.
- **Le nombre de fois où un processeur peut participer à un calcul corrompu** (*i.e.*, un calcul qui ne dépend pas causalement d'un cycle de calcul conforme aux spécifications). Par exemple, pour les circulations de jeton instantanément stabilisantes dans [CDPV06, CDV05b], cela correspond au nombre de jetons non-initiés qu'un processeur peut recevoir (cf. sections 5.4 et 5.5). Ces protocoles de circulation de jeton instantanément stabilisants assurent l'unicité du jeton dès le deuxième jeton initié. Classiquement, la circulation de jeton est utilisée pour résoudre

l'exclusion mutuelle : le jeton est vu comme un privilège et un processeur peut exécuter la section critique uniquement lorsqu'il détient un jeton. Donc, en utilisant un des protocoles de circulation de jeton précédents ([CDPV06, CDV05b]) où le nombre de jetons non-initiés qu'un processeur p peut recevoir est borné par n où n est le nombre de processeurs du réseau, nous pouvons assurer que p exécutera la section critique au plus $n+1$ fois de manière non-exclusive.

– **Le nombre de fois où un processeur peut participer à un calcul corrompu sans le détecter comme tel**. Lorqu'un processeur détecte qu'il a participé à un calcul erroné via un mécanisme de correction, il peut agir ou plus simplement stopper le calcul en cours afin de limiter l'influence de ce calcul sur le reste du réseau. Par exemple, le protocole de PIR instantanément stabilisant dans [CDV06b] garantit que chaque processeur peut recevoir au plus deux messages corrompus sans le détecter. Lorsqu'un message est détecté comme corrompu, le processeur m'émet pas d'accusé de réception pour ce message. Cette propriété est intéressante, par exemple, en cas d'utilisation de ce protocole de PIR pour faire une réinitialisation globale du système (*global reset*). En effet, classiquement, les protocoles de PIR sont utilisés pour faire une réinitialisation globale du système comme suit :

  1. L'initiateur diffuse un message d'arrêt aux autres processeurs.
  2. À partir de la réception de ce message, les processeurs bloquent tous leurs calculs sauf ceux correspondant à la réinitialisation.
  3. Les processeurs réinitialisent leurs variables juste avant d'émettre leur accusé réception puis se débloquent.

  Dans ce cas, limiter le nombre d'émissions d'accusés de réception erronés permet aussi de limiter les réinitialisations locales inutiles.

– **Le temps d'exécution nécessaire pour effectuer le premier cycle de calcul conforme aux spécifications**. Cette mesure représente le temps nécessaire pour exécuter le premier service conforme aux spécifications. Pour un protocole de service instantanément stabilisant, ce temps correspond au temps d'exécution du premier cycle (complet) de calcul, *i.e.*, le délai plus le temps d'exécution du cycle de calcul à partir d'une action de démarrage. Pour les protocoles de service auto-stabilisants, ce temps correspond au délai plus le temps d'exécution des cycles de calcul non conformes aux spécifications plus le temps d'exécution d'un cycle de calcul conforme aux spécifications.

– **Le temps d'exécution nécessaire pour effectuer les cycles de calcul suivants**. De manière générale, pour un protocole de service stabilisant (auto-stabilisation ou stabilisation instantanée), le pire des cas du temps d'exécution d'un cycle (complet) de calcul conforme aux spécifications correspond au pire des cas de l'exécution du premier cycle conforme aux spécifications. Par exemple, dans les protocoles à vagues, le premier cycle de calcul conforme aux spécifications permet de forcer la suppression des comportements dits "anormaux" (*i.e.*, les comportements non-initiés inutiles au calcul du service fourni par le protocole) dûs à un état initial quelconque. Suite à ce cycle de calcul, tous les autres cycles démarrent à partir d'un état *a priori* moins problématique. Ce type d'état global correspond généralement aux états de démarrage du système une fois que les comportements dûs aux fautes transitoires ont été supprimées. À partir d'un tel état global, le comportement du protocole se simplifie (par exemple, les actions de corrections ne sont plus jamais activables) et le temps d'exécution du cycle de calcul s'accélère. Par exemple, la complexité en temps du protocole de circulation de jeton en profondeur de [CDPV06] passe de $O(n^2)$ pas de calcul élémentaires pour le premier cycle à $O(n)$ pas de calcul élémentaires pour les suivants où n est le nombre de processeurs.

– **Le surcoût pour obtenir la propriété de stabilisation** (auto-stabilisation ou stabilisation instantanée). Ce surcoût peut être évalué en terme de temps et de mémoire. Pour le surcoût en mémoire, il suffit d'étudier le rapport entre l'occupation mémoire du protocole stabilisant et

l'occupation mémoire du protocole non-tolérant aux fautes le plus efficace en mémoire. Pour le surcoût en temps, nous évaluons le temps d'exécution d'un cycle de calcul du protocole stabilisant à partir d'un *état initial normal* (*i.e.*, un état global ne contenant aucun comportement "anormal", cf. définition 4.3.4, page 37). Ensuite, nous étudions le rapport entre la complexité obtenue et le temps d'exécution d'un cycle de calcul du protocole non-tolérant aux fautes le plus efficace en temps.

Complexité en espace. La complexité en espace d'un protocole s'évalue en terme de *nombre d'états* ou d'*occupation mémoire*.

- L'état local d'un processeur est défini par la valeur de ses variables. La complexité en nombre d'états correspond au nombre d'états locaux possibles que peuvent prendre les processeurs.
- En ce qui concerne l'occupation mémoire, on évalue le nombre de bits nécessaires pour stocker les variables internes des processeurs. L'occupation mémoire se déduit aisément en fonction du nombre d'états. En effet, l'occupation mémoire d'un processeur est égale à la somme des parties entières par excès du logarithme en base deux du nombre de valeurs possibles de chacunes des variables.

Obtenir des protocoles efficaces ne réclamant qu'un minimum de mémoire sur les processeurs constitue un enjeu important pour la recherche en stabilisation. De nombreux travaux, par exemple [Her91b, Her92, FD94, JB95, Joh97, BDPV99c, PV99], traitent de l'efficacité en espace des protocoles. Un autre intérêt de minimiser l'espace mémoire est la simplification des protocoles et de leur analyse. En effet, moins un protocole a d'états possibles, moins le nombre de cas à étudier est important. Il faut noter que pour certains problèmes, comme par exemple le PIR dans l'arbre non orienté (cf. [BDPV99c]), il n'est pas possible d'obtenir une solution à la fois optimale en complexité en temps et en occupation mémoire. Dans ce cas, il convient de trouver, au cas par cas, quel est le meilleur compromis possible entre l'espace et le temps. En effet, il a été montré que pour ce problème ([BDPV99c]), si on souhaite obtenir une solution restreinte en occupation mémoire, alors cette solution sera forcément coûteuse en complexité en temps. Inversement, si on souhaite obtenir une solution optimale en temps, alors la solution sera coûteuse en occupation mémoire.

Chapitre 4

Le Modèle

Dans ce chapitre, nous allons définir le cadre théorique de cette thèse. Pour cela, nous rappellerons, tout d'abord, la définition d'un système de transitions (section 4.1). À partir de cette définition, nous formaliserons la notion de système distribué (section 4.2). Enfin, nous présenterons le modèle à états : le modèle de calcul utilisé pour décrire nos protocoles (section 4.3).

4.1 Système de transitions

Un algorithme distribué est souvent modélisé comme une collection d'événements discrets. Chaque événement correspond à un changement atomique d'un état du système vers un autre. Ce type de système est généralement décrit à l'aide de *systèmes de transitions* (également appelés *automates*) constitués d'états et de transitions étiquetées entre ces états. Tel [Tel01] définit un système de transitions comme suit :

Définition 4.1.1 (Système de transitions) *Un* système de transitions *est un triplet $\mathcal{S} = (\mathcal{C},\mapsto,\mathcal{I})$ tel que :*
- *$\mathcal{C}$ est un ensemble d'états,*
- *$\mapsto$ est une relation binaire de $\mathcal{C}$ dans $\mathcal{C}$,*
- *$\mathcal{I}$ est un sous-ensemble d'états de $\mathcal{C}$ dit "initiaux".*

Remarque 4.1.1 *La relation $\mapsto$ étant un sous ensemble de $\mathcal{C} \times \mathcal{C}$, tout couple d'états $(\alpha,\beta) \in \mapsto$ sera noté dans la suite $\alpha \mapsto \beta$.*

Définition 4.1.2 (État terminal) *Soit $\mathcal{S} = (\mathcal{C},\mapsto,\mathcal{I})$ un système de transitions. $\forall \gamma \in \mathcal{C}$, γ est* terminal *si et seulement si il n'existe pas de transition $\gamma \mapsto \gamma'$ dans $\mapsto$.*

Définition 4.1.3 (Exécution) *Une* exécution *de $\mathcal{S}$ est une suite* maximale *$e = \gamma_0, \gamma_1, ..., \gamma_i, \gamma_{i+1}, ...$ telle que :*
- *$\gamma_0 \in \mathcal{I}$.*
- *Pour tout couple d'états de e : (γ_i, γ_{i+1}), la transition $\gamma_i \mapsto \gamma_{i+1}$ existe ($\gamma_i \mapsto \gamma_{i+1}$ sera alors appelé* mouvement *ou* pas de calcul*).*
- *Si e est de longueur finie alors le dernier état de e est un état terminal.*

La définition suivante formalise la notion d'*état accessible*. En fait, un état γ_j est *accessible* depuis un état γ_i s'il existe une exécution dans laquelle γ_i précéde γ_j.

Définition 4.1.4 (État accessible) *Un état γ_j est* accessible *à partir de γ_i, noté $\gamma_i \rightsquigarrow \gamma_j$, s'il existe une exécution e de $\mathcal{S}$ telle que $e = \gamma_0, \ldots, \gamma_i, \ldots, \gamma_j, \ldots$.*

4.2 Système distribué

Nous allons modéliser un système distribué en utilisant les notions de graphe et de système de transitions. Par la suite, nous utiliserons la terminologie relative aux graphes pour manipuler la topologie d'un système distribué. Les définitions des termes usuels issus de la théorie des graphes peuvent être trouvés dans le livre de Berge [Ber83] et celui de Aho et Ullman [AU92]. Nous rappelons, en annexe (annexe A), quelques définitions relatives aux graphes utilisées dans cette thèse.

4.2.1 Définition

Un système distribué est généralement représenté sous la forme d'un graphe non-orienté connexe $G = (V,E)$ tel que :

- V représente l'ensemble des processeurs où chaque processeur p est muni d'un système de transitions $\mathcal{S}_p = (\mathcal{C}_p,\mapsto_p,\mathcal{I}_p)$,
- E représente l'ensemble des canaux bidirectionnels reliant les processeurs.

Le comportement de chaque processeur p est décrit par le système de transitions $\mathcal{S}_p$ aussi appelé *algorithme local*. Le *protocole* (ou *algorithme distribué*) étant la collection de tous les algorithmes locaux. Enfin, il faut noter que dans le cas d'un système distribué stabilisant nous avons $\mathcal{C}_p = \mathcal{I}_p$, $\forall p \in V$ car un tel système peut démarrer à partir de n'importe quel état initial.

4.2.2 Classification des systèmes distribués

D'après [Lyn96], il est possible de classer les systèmes distribués suivant plusieurs critères :

- *Anonyme*. Un système distribué est dit *anonyme* si tous les processeurs sont strictement identiques et sont incapables de se distinguer de façon déterministe des processeurs ayant le même degré. Un système distribué est dit *non anonyme* si les processeurs peuvent se distinguer (par exemple en utilisant un identifiant unique). Dans le cas où un seul processeur se distingue des autres, le réseau est dit *enraciné* et le processeur distingué est appelé *racine* du réseau. Les protocoles présentés dans cette thèse fonctionnent sur des réseaux enracinés quelconques. Dans la suite, nous noterons *r* la racine du réseau.
- *Uniforme*. Un système distribué est dit *uniforme* si tous les processeurs exécutent le même algorithme et n'utilisent pas d'informations globales discriminantes telles que, par exemple, des identités. Dans le cas contraire, le système est dit *non uniforme*. Dans le cas où quelques processeurs diffèrent de tous les autres de part leurs algorithmes ou l'utilisation d'informations globales discriminantes, le système est dit *semi-uniforme*.

4.2.3 Notations

Dans la suite, nous allons utiliser des notations relatives aux systèmes distribués présentées ci-dessous.

Nous noterons $\mathcal{E}$ l'ensemble des exécutions possibles d'un protocole $\mathcal{P}$ dans le système $\mathcal{S}$. Nous noterons $\mathcal{C}$ l'ensemble des états d'un système. Nous noterons $\mathcal{E}_\alpha$ le sous-ensemble des exécutions de $\mathcal{E}$ commencant par l'état (global) α. Nous utiliserons le graphe G, l'ensemble V et l'ensemble E pour faire respectivement référence au réseau, aux n processeurs et aux m canaux. Nous dénoterons le diamètre du réseau par D et le degré du réseau (*i.e.*, la valeur maximale par parmi les degrés locaux Δ_p des processeurs p) par Δ.

Pour simplifier l'écriture des protocoles, nous nous refèrerons au canal $\{p,q\}$ dans l'algorithme de p par l'étiquette q. Nous supposerons que les étiquettes référant aux canaux sont stockées dans l'ensemble $Neig_p$, $\forall p \in V$. Cet ensemble sera supposé constant et localement ordonné par la relation

binaire $\prec_p$. Dans le cas d'un système distribué identifié, nous refèrerons à l'identité du processeur p par la constante Id_p.

4.3 Modèle à états

Il existe de nombreux modèles permettant de modéliser un système distribuée parmi lesquels le modèle *à états* et le modèle *à passage de messages*. Ce dernier étant le plus proche de la réalité. Les protocoles présentés dans cette thèse sont écrits dans le modèle à états. Le modèle à états a été introduit par Dijkstra dans [Dij74]. La particularité de ce modèle est que chaque processeur du réseau a un accès direct en lecture à l'état local de tous ses voisins. Dans ce modèle, l'envoi et la réception de messages sont simulés par des accès en lecture. Donc, l'*état local* d'un processeur est défini par la valeur de ses variables. Dans la suite, nous parlerons (par abus de langage) d'état pour référer à l'état local d'un processeur. Nous appellerons *configuration* l'état global du système, *i.e.*, le produit cartésien des états de tous les processeurs du système. Dans le modèle à états, l'algorithme local d'un processeur (aussi appelé *programme*) est décrit par un ensemble fini de variables partagées et un ensemble fini de *règles* (ou *actions*). Chaque règle est de la forme suivante :

$$< Etiquette > :: < Garde > \rightarrow < Traitement > .$$

L'*étiquette* permet d'identifier une règle, elle sert essentiellement à simplifier les explications et les preuves. La *garde* d'une règle dans le programme du processeur p est un prédicat booléen défini sur des variables de p et de ses voisins. Le *traitement* relatif à une règle dans le programme de p est une suite d'instructions qui modifient une ou plusieurs variables de p.

La règle R est dite *activable* dans une configuration donnée si et seulement si sa garde est vraie dans cette configuration. Par abus de langage, nous dirons qu'un processeur est activable lorsque l'une de ses règles est activable.

Dans le modèle à états, chaque mouvement d'une exécution se décompose en trois phases :

1. Tous les processeurs évaluent les gardes de leurs règles.
2. Un ordonnanceur, appelé *démon*, choisit un ou plusieurs processeurs parmi les processeurs activables.
3. Chaque processeur activable choisi exécute une de ses règles activables (cette règle est aussi choisie par le démon).

Les phases 1. et 3. sont exécutées de manière synchrone et atomique par les processeurs. L'asynchronisme du modèle vient donc du démon puisqu'il est susceptible de retarder l'exécution des règles activables d'un ou plusieurs processeurs.

Dans cette thèse, nous avons décidé d'affecter des priorités entre les règles d'un même algorithme afin de simplifier l'écriture des gardes de ces règles. Cette priorité suit l'ordre d'apparition des règles dans le texte de l'algorithme. La première règle dans le texte est la plus prioritaire. Inversement, la dernière règle dans le texte est la moins prioritaire. À partir de maintenant, lorsqu'un processeur activable sera choisi par le démon, la règle qu'il exécutera sera la règle la plus prioritaire parmi ses règles activables. Supposer des priorités sur les règles n'est pas restrictif par rapport au modèle initial. En effet, des priorités sur les règles peuvent être facilement obtenues en réécrivant celle-ci. Par exemple, supposons deux règles R_1 et R_2 de la forme suivante :

$$R_1 \ :: \ G_1 \ \rightarrow \ T_1$$

$$R_2 \ :: \ G_2 \ \rightarrow \ T_2$$

Pour que R_1 devienne plus prioritaire que R_2, il suffit de réécrire R_2 comme une nouvelle règle R'_2 de la forme suivante :

$$R'_2 \ :: \ \neg G_1 \ \wedge G_2 \ \rightarrow \ T_2$$

En ce qui concerne l'exécution concurrente de plusieurs protocoles, nous supposerons qu'ils s'exécutent de manière indépendante, *i.e.*, lorsqu'un processeur p est choisi par le démon, il exécute, dans le même mouvement et en fonction des priorités, une règle activable de chaque algorithme ayant des règles activables pour p.

4.3.1 Les démons

Le *démon* est un mécanisme global indépendant des processeurs qui ordonnance l'exécution en choisissant quels sont les processeurs activables qui vont exécuter une règle à chaque mouvement. Il existe plusieurs types de démon dans la littérature. De manière générale, un démon est défini en fonction de deux critères : la *répartition* de ses choix et son *équité*.

La répartition du choix du démon. La répartition du choix d'un démon définit combien de processeurs activables peuvent être choisis par le démon à chaque mouvement (tant que le système n'est pas dans une configuration terminale). Il existe un certain nombre d'hypothèses sur la répartition du démon dans la littérature des systèmes distribués. Les plus courantes sont les suivantes :

- Le démon est *central* (ou *séquentiel*). Dans ce cas, le démon ne choisit qu'un seul processeur activable à chaque mouvement.
- Le démon est *distribué*. Le démon distribué choisit un à plusieurs processeurs parmi ceux qui sont activables à chaque mouvement.
- Le démon est *synchrone* (ou *parallèle*). À chaque mouvement, le démon synchrone choisit tous les processeurs activables.

L'équité du démon. L'équité du démon donne les conditions nécessaires pour qu'un processeur activable soit obligatoirement choisi au cours d'une exécution. Les hypothèses les plus courantes sur l'équité du démon sont les suivantes :

- Le démon est *fortement équitable*. Toute exécution ordonnancée suivant un démon fortement équitable ne contient aucun suffixe infini dans lequel un processeur infiniment souvent activable n'exécute jamais de règle.
- Le démon est *faiblement équitable*. Toute exécution ordonnancée suivant un démon faiblement équitable ne contient aucun suffixe infini dans lequel un processeur continûment activable n'exécute jamais de règle.
- Le démon est *inéquitable*. Un démon n'ayant aucune hypothèse particulière sur son équité est dit inéquitable. Dans ce cas, le démon est simplement supposé *propre*, *i.e.*, tant qu'il existe des processeurs activables, il en choisit au moins un lors de chaque mouvement. De ce fait, un processeur est sûr d'être choisi lors du mouvement $\gamma \mapsto \gamma'$ si et seulement s'il est le seul processeur activable dans γ.

Bien entendu, dans le cas synchrone, le démon est fortement équitable : tout processeur activable est choisi dans le mouvement courant.

4.3.2 Les rondes

Deux unités de mesures sont classiquement utilisées pour évaluer les complexités en temps des protocoles écrits dans le modèle à états : le nombre de mouvements et le nombre de *rondes* (*round*). La

ronde permet d'évaluer le temps d'exécution par rapport aux processeurs les plus lents tout en tenant compte des règles. Le notion de ronde a tout d'abord été définie par Dolev, Israëli et Moran dans [DIM97]. Puis, cette définition a été remaniée par Cournier, Datta, Petit et Villain dans [CDPV02] pour prendre en compte la notion de *neutralisation de règles*.

Un processeur p subit une neutralisation durant le mouvement $\gamma \mapsto \gamma'$ si p est activable dans γ mais non activable dans γ' sans avoir exécuté aucune règle durant $\gamma \mapsto \gamma'$. En fait, la neutralisation de p correspond à la situation suivante : un ou plusieurs voisins de p exécutent une règle durant $\gamma \mapsto \gamma'$ et les changements occasionnés par l'exécution de ces règles rendent toutes les règles de p non activables dans γ'.

En utilisant la notion de neutralisation, la notion de ronde est définie comme suit ([CDPV02]) :

Définition 4.3.1 (Ronde) *Étant donnée une exécution e. La première ronde de e, notée e', est le préfixe minimal de e contenant, pour chaque processeur activable lors de la première configuration de e, l'exécution d'une de ses règles ou la neutralisation du processeur. Soit $e = e'e''$. La seconde ronde de e correspond à la première ronde de e'', etc...*

La ronde est utilisée pour évaluer le temps d'exécution par rapport aux processeurs les plus lents. Intuitivement, un processeur lent est un processeur continûment activable qui est retenu d'agir à cause des choix du démon (seul le démon peut empêcher un processeur activable d'agir). Or, lorsqu'un processeur subit une neutralisation au cours de la ronde, il ne peut plus être considéré comme un processeur lent : ce n'est pas le démon qui l'empêche d'agir, il ne souhaite tout simplement plus agir. Donc, le comportement de ce processeur après cette neutralisation ne doit pas être pris en compte pour décider de la fin de la ronde. La notion de *neutralisation de règles* permet donc d'affiner les mesures de complexité. D'ailleurs, les complexités en nombre de rondes obtenues avec la définition de [CDPV02] majorent celles obtenues avec la définition de [DIM97]. En effet, la définition de [DIM97] diffère uniquement par la non prise en compte la notion de neutralisation. Donc, la longueur des rondes (*i.e.*, le nombre de mouvements) avec la définition de [DIM97] est supérieure ou égale à la longueur des rondes avec la définition de [CDPV02]. Par exemple, supposons qu'au cours d'une ronde, un processeur subisse une neutralisation et devienne inactivable pour toujours. Avec la définition de [DIM97], la ronde est alors de longueur infinie. Ce n'est pas forcément le cas avec la définition de [CDPV02] : si le démon choisi pour les processeurs activables au cours du mouvement suivant, alors la ronde termine dans la configuration suivante.

Dans le modèle à états, la notion de ronde est liée à l'équité du démon. Par exemple, toutes les rondes d'une exécution sous un démon faiblement équitable sont finis en nombre de mouvements. En effet, par définition, un démon faiblement équitable choisit en un temps fini tout processeur continûment activable. Donc, pour prouver qu'un protocole résout une tâche spécifiée avec un démon faiblement équitable, il suffit de montrer qu'il exécute cette tâche en un nombre fini de rondes. En effet, les rondes contenant un nombre fini de mouvements, l'exécution de cette tâche sera elle-aussi finie en nombre de mouvements. En revanche, un démon inéquitable peut ne jamais choisir un processeur continûment activable lorsque celui n'est jamais le seul processeur activable. Donc, pour certain protocole si on suppose un démon inéquitable, il est possible de construire des exécutions ayant une ronde infinie. Ainsi, le schéma de preuve précédent n'est pas suffisant pour prouver un protocole sous un démon inéquitable. Le théorème suivant est un outil permettant de prouver qu'un protocole fonctionne sous un démon inéquitable :

Théorème 4.3.1 *Soit $\mathcal{T}$ une tâche et $\mathcal{SP}_{\mathcal{T}}$ une spécification de $\mathcal{T}$. Si toute exécution d'un protocole $\mathcal{P}$ sous un démon inéquitable contient un nombre fini de mouvements alors :*

$\mathcal{P}$ satisfait $\mathcal{SP}_{\mathcal{T}}$ sous un démon inéquitable si et seulement si $\mathcal{P}$ satisfait $\mathcal{SP}_{\mathcal{T}}$ sous un démon faiblement équitable.

Preuve. Soit e une exécution de $\mathcal{P}$ sous l'hypothèse d'un démon inéquitable. Par hypothèse, chaque ronde de e est finie en nombre de mouvements. D'après la définition 4.3.1, cela signifie que chaque processeur activable au cours de l'exécution e exécute une règle activable de $\mathcal{P}$ ou subit une neutralisation en un nombre fini de mouvements après être devenu activable. En particulier, cette propriété est vraie pour chaque processeur continûment activable au cours de exécution e. Donc, il n'existe pas de processeur continûment activable dans e qui n'exécute jamais de règle. Ainsi, e est aussi une exécution de $\mathcal{P}$ sous un démon faiblement équitable et, sans perte de généralité, l'ensemble des exécutions de $\mathcal{P}$ sous un démon inéquitable est égal à l'ensemble des exécutions de $\mathcal{P}$ sous un démon faiblement équitable. D'où, le théorème est vérifié. □

4.3.3 Modèle à états et protocoles de service instantanément stabilisants

Gestion des requêtes. Nous avons vu qu'un protocole de service initie, via une action de démarrage, un service suite à la demande d'un processeur initiateur. Cette demande est effectuée via une *requête* extérieure au protocole. Pour les protocoles de service non-tolérants aux fautes ainsi que la plupart des protocoles de service instantanément stabilisants, cette notion de requête, bien qu'essentielle, est implicite (*i.e.*, elle n'apparaît pas clairement dans le code des protocoles). Cependant, chaque action de démarrage est supposée être exécutée uniquement à la suite d'une requête. En ce qui concerne les protocoles de service auto-stabilisants nous avons vu que la notion de requête avait purement et simplement disparu car ces protocoles répètent les cycles de calcul à l'infini afin d'assurer la convergence du système (cf. section 3.2.2).

Dans cette thèse, nous avons choisi d'expliciter les requêtes dans le code de nos protocoles. Expliciter la requête permet de montrer que dans la plupart des cas, comme dans [CDPV04, CDV05b], les protocoles de service instantanément stabilisants finissent par atteindre une configuration terminale en cas d'absence totale de requêtes.

Puisque nos protocoles sont écrits dans le modèle à états, nous gérons les requêtes via une variable partagée $Request_i \in \{Wait, In, Out\}$[1] et définie comme entrée-sortie dans l'algorithme de chaque initiateur i. Ensuite, nous considérons que $Request_i = Wait$ signifie que l'initiateur i demande et "attend" le service. Lors de l'initiation du service demandé (*i.e.*, lorsque i exécute une action de démarrage relative au service demandé), $Request_i$ passe de $Wait$ à In pour signifier que la requête a été prise en compte par le système. Finalement, $Request_i$ passe de In à Out lorsque i décidera que le service demandé a été effectué. Donc, le système sera prêt à exécuter une nouvelle requête à partir du processeur i uniquement lorsque la variable $Request_i$ sera (à nouveau) égale à Out. Naturellement, les transitions de $Wait$ à In et de In à Out sont gérées dans le code du protocole même, tandis que la transition de Out à $Wait$ est gérée de manière externe (*i.e.*, cela signifie qu'un autre protocole, appelé application, doit effectuer cette transition). Il est important de noter que seules les transitions citées précédemment sont autorisées : toutes les autres transitions, par exemple la transition de In vers $Wait$, sont interdites. Dans la suite, nous représenterons l'action correspondant à la requête externe par la règle IR[2] (cette règle est définie pour chaque initiateur i) :

$$IR :: ApplicationRequest(i) \wedge (Request_i = Out) \rightarrow Request_i := Wait; ApplicationRelease_i;$$

Dans cette règle, $ApplicationRequest(i)$ est un prédicat qui est vrai lorsqu'une application de l'initiateur i demande un service. $ApplicationRelease_i$ est une fonction qui contient le code que l'application doit exécuter lorsque la requête est prise en compte par le système. En particulier, $ApplicationRequest(i)$ doit devenir faux suite à l'exécution de $ApplicationRelease_i$. Par la suite, nous

[1] Pour éviter toute ambiguïté, cette variable pourra être notée $\mathcal{P}.Request_i$ pour faire référence à la variable $Request_i$ spécifique au protocole $\mathcal{P}$.

[2] Signifie *Interface Request*.

supposerons que, dès qu'il est satisfait, le prédicat $ApplicationRequest(i)$ reste vrai continûment jusqu'à ce que la règle IR soit exécutée.

Configurations initiales normales d'un protocole mono-initiateur. La notion de légitimité n'est pas discriminatoire en stabilisation instantanée : comme le système vérifie toujours ses spécifications, toute configuration est considérée comme légitime. Cependant, pour les protocoles de service instantanément stabilisants, nous distinguerons souvent un sous-ensemble de configurations dit *ensemble des configurations initiales normales* (cette distinction est aussi valable pour les protocoles de service auto-stabilisant). En fait, les configurations initiales normales correspondent aux configurations de démarrage du système une fois que les comportements "anormaux" dûs aux fautes transitoires ont été supprimées. Cette notion permet notamment d'étudier le surcoût de la stabilisation. Nous définissons maintenant la notion de configuration initiale normale d'un protocole de service mono-initiateur. Cette notion fait appel deux autres concepts définis ci-dessous : les *configurations originelles* et les *configurations normales*.

Les configurations originelles d'un système correspondent à l'initialisation des variables du protocole dans un environnement sans panne avant la première exécution. Dans ce cas, tous les processeurs sont en attente du démarrage de l'exécution, démarrage qui sera assuré par l'initiateur lorsque celui-ci recevra une requête.

L'ensemble des configurations normales représente l'ensemble des configurations possibles dans une exécution n'ayant jamais subi de fautes.

L'ensemble des configurations initiales normales ne se limite *a priori* pas aux configurations de l'ensemble des configurations originelles. En effet, en régime normal, un protocole mono-initiateur peut démarrer un nouveau cycle de calcul à partir d'une configuration dans laquelle des reliquats du cycle de calcul précédent perdurent dans le réseau. Or, comme ces reliquats ne sont pas consécutifs à un comportement anormal du système, cette configuration initiale doit être considérée comme normale (nous verrons un exemple de ce type de configuration dans la sous-section 5.4.1).

Définition 4.3.2 (Configuration originelle) *Nous appelons* ensemble des configurations originelles *d'un protocole de service mono-initiateur $\mathcal{P}$ le sous-ensemble de configurations γ_i vérifiant :*

1. *$\forall p \in V$, p n'est pas activable dans γ_i.*
2. *L'initiateur p vérifie $Request_p = Out$ (*i.e.*, p est prêt à recevoir une requête) dans γ_i.*
3. *L'initiateur p est activable dans γ_{i+1} si et seulement si p reçoit une requête dans $\gamma_i \mapsto \gamma_{i+1}$ (*i.e.*, la règle IR de p est exécutée dans $\gamma_i \mapsto \gamma_{i+1}$).*

Définition 4.3.3 (Configuration normale) *Une* configuration normale *du protocole de service mono-initiateur $\mathcal{P}$ est une configuration accessible à partir d'une configuration originelle (les configurations originelles incluses).*

Définition 4.3.4 (Configuration initiale normale) *Une* configuration initiale normale *du protocole de service mono-initiateur $\mathcal{P}$ est une configuration normale de $\mathcal{P}$, γ_i, où l'initiateur p vérifie :*

- *p n'est pas activable dans γ_i, et*
- *p est activable dans γ_{i+1} si et seulement si p reçoit une requête dans $\gamma_i \mapsto \gamma_{i+1}$.*

Deuxième partie

Contributions

Chapitre 5

Parcours en profondeur instantanément stabilisants

Le *parcours en profondeur* (DFS : *Depth-First Search*) est le parcours séquentiel le plus classique de la littérature [Tar72]. Le parcours en profondeur a de nombreuses applications. Dans les systèmes distribués, il est principalement utilisé pour résoudre le problème d'exclusion mutuelle. Cependant, il peut être utilisé pour résoudre de nombreuses autres tâches : par exemple, le calcul d'arbre couvrant ([HC93]), la programmation par contrainte ([JSYZ04]), le routage par intervalle ([vLT87]) ou pour évaluer des propriétés globales sur le réseau telles que les points d'articulation (cf. [Tar72] et section 5.7).

Dans ce chapitre, nous allons tout d'abord définir formellement le concept de parcours en profondeur (section 5.1). Nous proposons ensuite un état de l'art non exhaustif sur les parcours en profondeur distribués (section 5.2). Puis, nous présentons deux protocoles basiques de parcours en profondeur non-tolérants aux fautes, l'un dans le modèle à passage de messages et l'autre dans le modèle à états. Nos solutions instantanément stabilisantes sont présentées dans les sections 5.4 et 5.5. Nos deux parcours en profondeur sont des protocoles instantanément stabilisants écrits dans le modèle à états (cf. section 4.3). Le premier (section 5.4) est basé sur des listes d'identités. Le second (section 5.5) utilise un principe de question/réponse pour remplacer les listes d'identités. Il faut noter que ces deux solutions fonctionnent sous l'hypothèse d'un démon distribué inéquitable : le démon le plus général du modèle à états. Comme expliqué dans la sous-section 4.3.3, le code de ces deux protocoles fera explicitement référence aux requêtes via la variable partagée $Request$. Cependant, pour simplifier l'analyse des protocoles, nous ne tiendrons pas compte, dans un premier temps, de cette variable $Request$ dans les preuves. Nous traiterons la gestion explicite de la requête dans une section indépendante (cf. section 5.6) et notamment des répercussions au niveau de la complexité en temps. Enfin, nous présenterons dans la section 5.7 deux applications instantanément stabilisantes pouvant être obtenues à partir de nos deux protocoles de parcours en profondeur. Ces deux applications évaluent des propriétés globales sur le réseau. La première application (cf. sous-section 5.7.3) est un calcul de point fixe avec détection de terminaison. L'algorithme présenté permet de marquer les *points d'articulation* (définition A.2.15, page 161) et les *isthmes* (définition A.2.16, page 161) du réseau. Les points d'articulation (resp. les isthmes) sont des processeurs (resp. des canaux) dont la suppression (suite à une panne définitive, par exemple) provoque la partition du réseau en plusieurs composantes connexes. De plus, l'existence de points d'articulation ou d'isthmes dans le réseau peut être l'une des causes d'apparition de congestions. L'identification des points d'articulation et des isthmes est donc essentielle du point de vue de la tolérance aux fautes. La seconde application (cf. sous-section 5.7.4) permet d'évaluer si un ensemble fourni par l'application est un *ensemble séparateur* (définition A.2.14, page 161) du réseau. Un ensemble séparateur (*cutset*) est un sous-ensemble de processeurs du réseau dont la suppression (suite à des pannes définitives, par exemple) provoque la partition du

réseau en plusieurs composantes connexes. La détection d'ensembles séparateurs est un problème important dans de nombreuses applications telles que l'évaluation de la fiabilité des réseaux.

5.1 Définition

Le concept de parcours en profondeur est basé sur la notion de parcours séquentiel. Donc, avant de définir le parcours en profondeur (sous-section 5.1.2), nous allons définir la notion de parcours séquentiel (sous-section 5.1.1).

5.1.1 Protocoles de parcours séquentiel

Les protocoles de parcours en profondeur font partie d'une sous-classe des protocoles à vagues (définis page 21) : la classe des protocoles de *parcours séquentiels*. En algorithmique distribuée, ce type de protocole utilise le plus souvent un objet particulier appelé *jeton*. Les particularités d'un protocole de parcours séquentiel $\mathcal{P}$ par rapport aux autres protocoles à vagues sont les suivantes [Tel01] :

1. Chaque vague de $\mathcal{P}$ a un seul initiateur i qui initie le calcul par une seule action de démarrage. Cette action déclenche une action sur un et un seul voisin de i.
2. Pour tout processeur p, pour toute action a autre que l'action de démarrage, soit a est destinée à déclencher une action sur un et un seul voisin de p, soit a est une action de décision.

Ces deux propriétés impliquent que dans chaque vague exactement un processeur décide. Nous disons que le protocole se termine au processeur q lorsque q décide. La dernière particularité d'un protocole de parcours séquentiel $\mathcal{P}$ est la suivante :

3. Chaque vague de $\mathcal{P}$ se termine à l'initiateur.

Les protocoles de parcours séquentiel sont donc des protocoles à vagues à un seul initiateur et dont le degré de parallélisation est 1. Il faut noter que, mis à part le parcours en profondeur, il existe d'autres types de parcours. Par exemple, dans [Cha82, Seg83], les auteurs proposent (entre autres) deux protocoles qui parcourent le réseau dans un ordre quelconque.

5.1.2 Parcours en profondeur

Le problème du parcours en profondeur est le suivant : à partir de l'initiateur, il faut parcourir (ou visiter) l'ensemble des processeurs du réseau de façon séquentielle (généralement en utilisant un jeton) et le parcours doit s'acheminer en priorité via les processeurs non encore visités. Nous allons maintenant donner une définition formelle du *parcours en profondeur* (spécification 5.1.1). Pour cela, nous allons notamment utiliser la notion de *v-arête* (définition 5.1.1).

Définition 5.1.1 (V-arête) *Soit $\mathcal{L} = \{p_0,p_1\}$, ..., $\{p_i,p_{i+1}\}$, ..., $\{p_{k-1},p_k\}$ la suite des arêtes traversées lors d'un parcours. Nous appelons* v-arête *de p_i la première arête de $\mathcal{L}$ dont p_i est incident.*

Spécification 5.1.1 (Parcours en profondeur) *Soit $\mathcal{L} = \{p_0,p_1\}$, ..., $\{p_i,p_{i+1}\}$, ..., $\{p_{k-1},p_k\}$ la suite des arêtes traversées lors d'un parcours séquentiel. Soit E_α l'ensemble de toutes les v-arêtes de $\mathcal{L}$. Le parcours associé à $\mathcal{L}$ est un* parcours en profondeur *effectué à partir de p_0 si et seulement si le graphe partiel $G_\alpha(p_0) = (V,E_\alpha)$ est un arbre couvrant en profondeur (cf. définition A.2.12, page 161).*

Nous allons maintenant présenter un état de l'art (non exhaustif) des protocoles de parcours en profondeur distribué de la littérature.

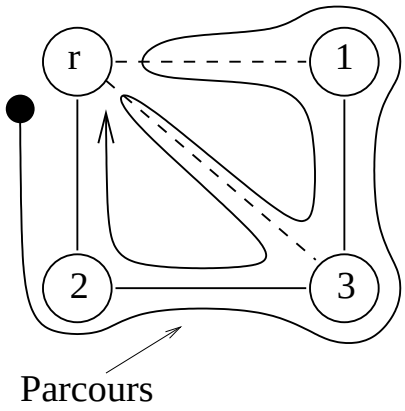


FIG. 5.1.1 – Un exemple de parcours en profondeur.

5.2 État de l'art

De nombreux protocoles de parcours en profondeur non tolérants aux fautes ont été proposés dans la littérature, par exemple, [Cha82, Che83, Awe85b, Cid88]. Ces protocoles ont été écrits dans le modèle à passage de messages (ce modèle est plus proche de la réalité que le modèle à états). Dans le domaine des systèmes auto-stabilisants, le problème du parcours en profondeur auto-stabilisant a tout d'abord été considéré par Huang et Chen dans [HC93]. Cette solution utilise $\Omega(\Delta \times n)$ états (*i.e.*, $\Omega(\log \Delta + \log n)$ bits) par processeur où n est le nombre de processeurs et Δ le degré du réseau. Par la suite, de nombreuses solutions auto-stabilisantes ont été présentées, par exemple, [JB95, JABD97, DJPV00]. Dans tous ces articles ([HC93, JB95, JABD97, DJPV00]), les protocoles présentés sont écrits dans le modèle à états et ont un temps de stabilisation en $\Omega(n \times D)$ rondes dans le pire des cas (où D est le diamètre du réseau). Dans [JB95, JABD97, DJPV00], les auteurs améliorent l'occupation mémoire de [HC93] en passant de $O(\log n + \log \Delta)$ à $O(\log \Delta)$ bits par processeur. Le protocole de [DJPV00] offre la meilleure complexité en espace : $3(\Delta_p + 1)$ pour chaque processeur $p \neq r$ et $2(\Delta_r + 1)$ pour r. Récemment, Petit a proposé dans [Pet01] un protocole de parcours en profondeur auto-stabilisant dont le temps de stabilisation est en $O(n)$ rondes. Cependant, toutes les solutions auto-stabilisantes proposées sont prouvées sous l'hypothèse d'un démon faiblement équitable. Finalement, il faut noter que Petit et Villain ont proposé une solution auto-stabilisante dans le modèle à messages [PV97]. La première solution instantanément stabilisante pour ce problème a été proposée dans [PV99] mais pour des réseaux en arbres. Pour les réseaux enracinés quelconques (excepté nos solutions), il faut noter que le "transformateur" proposé dans [CDPV03] (écrit dans le modèle à états) permet de construire un protocole instantanément stabilisant de parcours en profondeur à partir d'un protocole construit pour un environnement sans faute. Cependant, avec ce transformateur, les solutions obtenues ne fonctionnent qu'avec un démon faiblement équitable et peuvent être très coûteuses en temps d'exécution. En effet, ce transformateur génère des calculs d'état global du système pour évaluer régulièrement un prédicat défini sur les variables du protocole à transformer. Or, le nombre de calculs d'état global généré par le transformateur est indépendant du temps d'exécution du protocole initial.

Nous présentons maintenant deux protocoles simples de parcours en profondeur, l'un dans le modèle à passage de messages, l'autre dans le modèle à états.

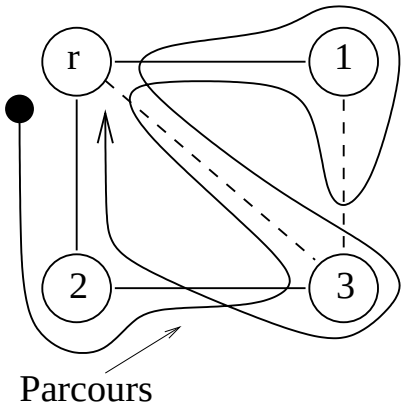


FIG. 5.3.2 – Un exemple de parcours quelconque.

5.3 Deux exemples de protocoles non-tolérants aux fautes.

Dans le modèle à passage de messages, le protocole basique de parcours en profondeur ([Cha82]) fonctionne comme suit. Au début d'un parcours en profondeur, l'initiateur crée un jeton et marque les canaux de tous ses voisins comme "non visité". Puis, il choisit un voisin comme *successeur* courant dans le parcours, lui envoie le jeton et marque son canal comme "visité".

Lorsqu'un processeur — y compris l'initiateur — reçoit le jeton, celui-ci marque le canal de l'émetteur comme "visité". Ensuite, s'il s'agit de la première réception du jeton alors le processeur repère l'émetteur du jeton comme son *père* dans le parcours et marque les canaux de tous ses voisins sauf celui de son père comme "non visité". Puis, dans tous les cas, le processeur exécute les actions suivantes :

- Soit tous les canaux du processeur sont marqués "visité" et, dans ce cas, si le processeur est l'initiateur du parcours alors il décide (et le parcours est terminé) sinon il renvoie le jeton à son père.
- Soit le processeur a des canaux qui sont marqués "non visité". Dans ce cas, si le processeur n'a pas de successeur courant (c'est la première fois qu'il reçoit le jeton) ou si l'émetteur du jeton est son successeur courant alors il choisit un (nouveau) successeur courant parmi ses voisins dont le canal est marqué "non visité", lui envoie le jeton et marque son canal comme "visité". Dans le cas contraire, le processeur renvoie le jeton à l'émetteur.

La figure 5.1.1 nous montre un parcours en profondeur à partir du processeur *r* pouvant être exécuté avec le protocole précédent. Dans cette figure, les v-arêtes sont représentées par les lignes pleines et nous pouvons remarquer qu'elles définisssent un arbre couvrant en profondeur enraciné en *r*. Au contraire, le parcours présenté dans la figure 5.3.2 n'est pas un parcours en profondeur. En effet, il n'y a, par exemple, pas de relation ancêtre/descendant entre les deux voisins 1 et 3, donc, l'arbre défini par les v-arêtes ne vérifie pas la définition A.2.12 (page 161) : ce n'est pas un arbre couvrant en profondeur.

Dans cette thèse, nous nous sommes intéressés au parcours en profondeur dans un réseau enraciné quelconque où la racine, *r*, correspond toujours à l'initiateur du parcours. Ensuite, nos protocoles utilisent une méthode classique des systèmes distribués pour parcourir le réseau en profondeur d'abord. Cette méthode est présentée dans la remarque suivante :

Remarque 5.3.1 *Classiquement, dans le modèle à états, un parcours en profondeur est exécuté en utilisant un jeton. À partir de la racine (*r*), le jeton est passé de voisin en voisin via des processeurs non visités jusqu'à ce qu'il atteigne un processeur ayant tous ses voisins visités. À partir de là, le*

jeton remonte dans l'arbre couvrant en construction via les liens pères jusqu'à ce qu'il atteigne un processeur p vérifiant l'un de ses deux cas :

- *p a des voisins non visité. Dans ce cas, le jeton est passé à un voisin non visité de p et le parcours reprend comme précédemment.*
- *$p =$* r *et tous les voisins de p sont visités. Dans ce cas,* r *décide de la terminaison du parcours.*

Suivant la méthode présentée dans la remarque 5.3.1, nous présentons maintenant un protocole de parcours en profondeur simple conçu pour un environnement sans faute et écrit dans le modèle à états. Ce protocole, appelé $\mathcal{DFS}0$, fonctionne dans des réseaux enracinés de topologie quelconque. Il est presenté dans les algorithmes 5.3.1 et 5.3.2.

Le protocole $\mathcal{DFS}0$ est divisé en deux phases :

- La *phase de visite* où un jeton visite les processeurs en profondeur d'abord (cette phase est aussi appelée *circulation de jeton*).
- La *phase de nettoyage* où les traces du parcours sont effacées pour permettre à la racine d'initier une autre circulation de jeton.

Pour réaliser ces deux phases, chaque processeur p du protocole maintient les deux variables suivantes :

- $S_p \in Neig_p \cup \{C,D\}$. Initialement, la variable S_p est égale à C (*Clean*) pour signifier que p n'est traversé par aucun parcours : p attend d'être visité par un nouveau parcours (*i.e.,* il attend de recevoir le jeton pour la première fois). Lorsque $S_p \in Neig_p$, cela signifie que p participe au parcours courant et désigne le processeur S_p comme un successeur courant de p dans le parcours. Enfin, $S_p = D$ (*Done*) signifie qu'un parcours est passé par p, que les visites de ce parcours à partir de p sont terminées et que p est en attente de nettoyage.
- Par_p. $Par_p \in Neig_p$ pour $p \neq r$ et $Par_p = \perp$ pour $p = r$. Par_p est utilisé pour pointer le père de p dans le parcours courant, *i.e.,* le voisin qui lui a envoyé la première fois le jeton. Les variables Par permettent de garder une trace de l'arbre en profondeur suivi par le parcours à partir de la racine (les variables Par ne sont jamais réinitialisées). Par définition, *r* n'a jamais de père dans un parcours (*r* est l'initiateur), donc Par_r est une constante égale à $\perp$. Contrairement à la variable S_p, la variable Par_p n'est pas essentielle pour effectuer un parcours en profondeur, par exemple, un protocole de parcours en profondeur non-tolérant aux fautes et n'utilisant pas de pointeur père est présenté dans la thèse de Petit [Pet98]. Cependant, nous avons choisi d'utiliser le pointeur père pour avoir une phase de visite similaire aux protocoles instantanément stabilisants de parcours en profondeur présentés dans la suite de cette thèse.

Décrivons maintenant, à partir de l'exemple fourni figure 5.3.3, le comportement de notre protocole $\mathcal{DFS}0$ (algorithmes 5.3.1 et 5.3.2). Initialement, le système est dans une configuration où $(Request_r = Out) \wedge (\forall p \in V,\ S_p = C)$. Supposons que la racine, *r*, reçoive une requête (*i.e.,* $Request_r := Wait$) via l'exécution de la règle externe IR. Suite à cette requête, le système se retrouve dans une configuration similaire à la configuration (i) de la figure 5.3.3. Dans cette configuration, l'action de démarrage, *i.e.,* la règle F de *r*, est l'unique règle activable du système. *r* exécute donc sa règle F dans le premier mouvement ($(i) \mapsto (ii)$) : *r* amorce une phase de visite en désignant comme successeur son voisin minimal dans l'ordre local $\prec_r$ (*i.e., r* crée un jeton et l'envoie à ce voisin) et en affectant In à $Request_r$ pour signifier à l'application que sa requête a été prise en compte. Dans le mouvement suivant ($(ii) \mapsto (iii)$), le successeur désigné p pointe son père (*r*) avec sa variable Par_p et prolonge la phase de visite en désignant comme successeur son voisin non visité q minimal dans l'ordre local $\prec_p$: p reçoit le jeton de *r* puis l'envoie à q (règle F). La phase progresse ainsi séquentiellement dans la profondeur du réseau jusqu'à atteindre un processeur q n'ayant plus aucun voisin non visité (configuration iv). Lorsqu'il est atteint, q pointe son père et affecte la valeur D à sa variable S_q : il renvoie le jeton à son père pour que celui-ci reprenne le parcours en changeant

Algorithm 5.3.1 Algorithme $\mathcal{DFS}0$ pour $p = r$

Entrée-sortie : $Request_p \in \{Wait, In, Out\}$ initialisé à Out ;

Entrée : $Neig_p$: ensemble des voisins (localement ordonné) ;

Constante : $Par_p = \perp$;

Variable : $S_p \in Neig_p \cup \{C, D\}$ initialisé à C ;

Macro :

$Next_p \quad = \quad (q = \min_{\prec_p}\{q' \in Neig_p :: S_{q'} = C\})$ **si** q **existe**, D **sinon** ;

Prédicats :

$Leaf(p) \quad \equiv \quad (\forall q \in Neig_p :: (Par_q = p) \Rightarrow (S_q = C))$

$Forward(p) \quad \equiv \quad (S_p = C)$

$Backward(p) \quad \equiv \quad (\exists q \in Neig_p :: S_p = q \wedge S_q = D)$

$Clean(p) \quad \equiv \quad (S_p = D) \wedge Leaf(p)$

Règles :

Phase de Visite :

$F :: Forward(p) \wedge (Request_p = Wait) \rightarrow S_p := Next_p ; Request_p := In ;$

$B :: Backward(p) \rightarrow S_p := Next_p ;$

Phase de Nettoyage :

$C :: Clean(p) \rightarrow S_p := C ; Request_p := Out ;$

son successeur courant (règle F dans le mouvement $(iv) \mapsto (v)$). Dans le mouvement suivant, le père de q affecte lui-aussi la valeur D à sa variable S_q (il renvoie le jeton à son père) car il n'a plus de voisin non visité (règle B). Le jeton remonte ainsi en suivant les pointeurs pères jusqu'à atteindre un processeur ayant encore des voisins non visités (le processeur *r* dans la configuration vii). Le processeur désigne alors un nouveau successeur parmi ses voisins non visités et lui envoie le jeton en exécutant sa règle B (cf. *r* dans le mouvement $(vii) \mapsto (viii)$). Ainsi, la phase de visite reprend à partir de ce nouveau succcesseur. En parallèle de la remontée du jeton, la phase de nettoyage s'amorce. Le nettoyage est effectué à partir des feuilles de l'arbre couvrant calculé (règle C). Un processeur $p \neq r$ peut se nettoyer lorsque tout son sous-arbre a été nettoyé et que plus aucun de ses voisins ne participe à la phase de visite ($\forall q \in Neig_p$, $S_q \in \{C, D\}$). Par ce mécanisme, la phase de visite finit par terminer à la racine (mouvement $(ix) \mapsto (x)$). La phase de nettoyage se termine par la suite à la racine (mouvement $(xi) \mapsto (xii)$) : la racine se nettoie, via la règle C, après que tous ses voisins se sont nettoyés (mouvement $(xi) \mapsto (xii)$). La règle C de la racine est aussi utilisée pour signifier à l'application que le parcours demandé est terminé et que le système est à nouveau en attente de requête (*i.e.*, $Request_r := Out$).

Le protocole $\mathcal{DFS}0$ (cf. algorithmes 5.3.1 et 5.3.2) est asymptotiquement optimal en temps d'exécution ($O(n)$ mouvements). Il faut noter que la borne inférieure du temps d'exécution de tout parcours en profondeur dans un réseau enraciné quelconque conçu pour un environnement sans faute et écrit dans le modèle à états est exactement $2(n-1)$ mouvements (le nombre de mouvements nécessaires pour parcourir les arêtes de l'arbre couvrant dans les deux sens) et $2(n-1)$ rondes (le parcours est séquentiel, *i.e.* son degré de parallélisation est 1). Un protocole ayant exactement cette complexité en temps d'exécution peut être facilement réalisé : il suffit de remplacer la phase de nettoyage par un mécanisme utilisant deux couleurs (un tel protocole peut être trouvé dans la thèse de Petit, [Pet98]). Cependant, nous avons choisi de présenter le protocole $\mathcal{DFS}0$ car sa phase de visite sera réutilisée de manière quasi identique dans les protocoles instantanément stabilisants de parcours en profondeur présentés dans la suite de cette thèse.

Nous allons maintenant présenter deux solutions instantanément stabilisantes pour le problème de parcours en profondeur dans un réseau enraciné quelconque. Ces deux solutions ont la particularité de fonctionner avec un démon distribué inéquitable. La première solution est basée sur des listes d'identités. La seconde utilise un principe de question/réponse pour remplacer les listes d'identités.

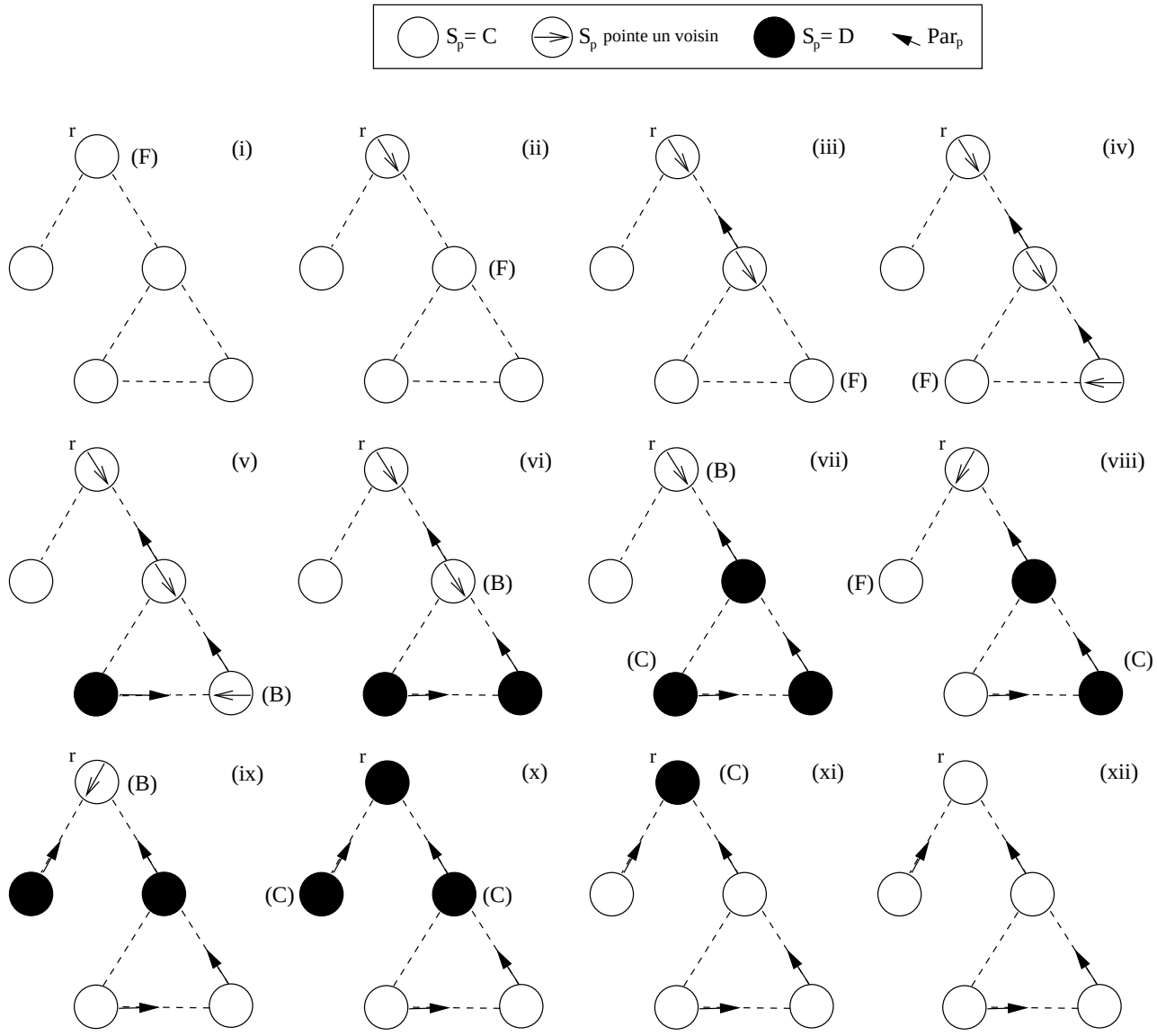


FIG. 5.3.3 – Exemple d'exécution de $\mathcal{DFS}0$.

5.4 Première solution

La première solution que nous proposons est basée sur l'utilisation de listes d'identités. Chaque processeur maintient une liste d'identités durant le parcours. Ces listes d'identités représentent une mémoire des processeurs visités par le parcours courant. Durant un parcours initié à la racine, nous maintenons l'invariant suivant : la liste du processeur qui détient le jeton contient les identités de tous les processeurs visités par le jeton. Cette solution a donné lieu à deux publications [CDPV04, CDPV06].

Nous allons maintenant présenter informellement notre protocole, appelé protocole $\mathcal{DFS}1$, basé sur des listes d'identités (sous-section 5.4.1). Ensuite, nous proposerons une preuve de la stabilisation instantanée de ce protocole (sous-section 5.4.2). Puis, nous discuterons de la complexité du protocole (sous-section 5.4.3) avant de conclure (sous-section 5.4.4).

Algorithm 5.3.2 Algorithme $\mathcal{DFS}0$ pour $p \neq r$

Entrée : $Neig_p$: ensemble des voisins (localement ordonné) ;

Variables : $S_p \in Neig_p \cup \{C,D\}$ initialisé à C ; $Par_p \in Neig_p$ (non initialisé) ;

Macros :

$Pred_p \quad = \quad \{q \in Neig_p :: S_q = p\}$;

$Next_p \quad = \quad (q = \min_{\prec_p}\{q' \in Neig_p :: S_{q'} = C\})$ **si** q **existe**, D **sinon** ;

Prédicats :

$Leaf(p) \quad \equiv \quad (\forall q \in Neig_p :: (Par_q = p) \Rightarrow (S_q = C))$

$Forward(p) \quad \equiv \quad (|Pred_p| = 1) \wedge (S_p = C)$

$Backward(p) \quad \equiv \quad (\exists q \in Neig_p :: S_p = q \wedge S_q = D)$

$Clean(p) \quad \equiv \quad (S_p = D) \wedge Leaf(p) \wedge (\forall q \in Neig_p :: S_q \in \{C,D\})$

Règles :

Phase de Visite :

$F :: Forward(p) \rightarrow S_p := Next_p ; Par_p := (q \in Pred_p)$;

$B :: Backward(p) \rightarrow S_p := Next_p$;

Phase de Nettoyage :

$C :: Clean(p) \rightarrow S_p := C$;

5.4.1 Le protocole

Pour expliquer le fonctionnement du protocole $\mathcal{DFS}1$, nous allons, tout d'abord, présenter son principe de base. Puis, nous présenterons son comportement à partir d'une configuration initiale normale. Enfin, nous expliquerons les mécanismes supplémentaires utilisés pour assurer son fonctionnement correct lorsqu'il démarre à partir d'une configuration initiale quelconque (*phase de correction*).

Principe. Le protocole $\mathcal{DFS}1$ reprend le principe des deux phases du protocole $\mathcal{DFS}0$ (*i.e.*, la *phase de visite* et la *phase de nettoyage*).

En fonctionnement normal, la phase de visite s'effectue comme dans le protocole $\mathcal{DFS}0$. Cependant, à chaque première visite du jeton sur un processeur (règle F), ce processeur ajoute son identité dans la liste des processeurs visités (cette liste sera notée $Visited$). Une conséquence de cette option est de simplifier la phase de nettoyage : nous pouvons autoriser un processeur à se nettoyer dès lors qu'il n'est plus pointé par son père puisqu'en figurant dans la liste il ne sera pas confondu avec un processeur non visité. La contrepartie est que lors d'un démarrage de la racine, le système peut contenir des processeurs qui ont encore leur variable S à D.

En démarrant d'une configuration quelconque, le système peut contenir plusieurs jetons, chacun correspondant à un parcours particulier. Lorsque le jeton du parcours initié par *r* rencontre un de ces parcours anormaux, le processeur détenant ce jeton peut s'apercevoir de l'absence dans sa liste du voisin appartenant à un parcours anormal. Il attendra donc que ce voisin se nettoie grâce à la *phase de correction* (présentée par la suite) afin de pouvoir l'intégrer à son propre parcours.

Dans le paragraphe suivant, nous donnons un exemple de fonctionnement du protocole $\mathcal{DFS}1$ à partir d'une configuration initiale normale en nous basant sur les figures 5.4.4 et 5.4.5. Dans ces deux figures (et pour toutes les figures présentées dans la suite de cette thèse), nous montrons les valeurs des variables uniquement lorsqu'elles ont une incidence sur le comportement du protocole.

Fonctionnement normal. Nous considérons comme configuration initiale normale du protocole $\mathcal{DFS}1$ toute configuration telle que $(S_r = C \wedge Request_r = Out) \wedge (\forall\, p \in Neig_r, S_p = C) \wedge (\forall\, q \in V \setminus Neig_r, S_q \in \{C,D\})$. Supposons que le système soit dans une configuration initiale normale et que la racine du réseau *r* reçoive une requête (*i.e.*, $Request_r := Wait$) par la règle externe IR. Suite à cette requête, le système se retrouve dans une configuration similaire à la configuration i de la figure 5.4.4. Dans cette configuration, les processeurs 2, 3 et 5 ont leur règle C activable et la racine, *r*, a sa règle F activable. La règle C permet à un processeur de réaliser sa phase de nettoyage. Pour

Algorithm 5.4.3 Algorithme $\mathcal{DFS}1$ pour $p = r$

Entrée-sortie : $Request_p \in \{Wait, In, Out\}$;

Entrées : $Neig_p$: ensemble des voisins (localement ordonnés) ; Id_p : identité de p ;

Constante : $Par_p = \perp$;

Variables : $S_p \in Neig_p \cup \{C, D\}$; $Visited_p$: ensemble d'identités ;

Macros :

$Next_p = (q = \min_{\prec_p}\{q' \in Neig_p :: (Id_{q'} \notin Visited_p)\})$ **si** q **existe**, D **sinon** ;

$ChildVisited_p = Visited_{S_p}$ **si** $(S_p \notin \{C, D\})$, $\emptyset$ **sinon** ;

Prédicats :

$SetError(p) \equiv (S_p \neq C) \wedge [(Id_p \notin Visited_p) \vee (\exists q \in Neig_p :: (S_p = q) \wedge (Id_q \in Visited_p))]$

$Error(p) \equiv SetError(p)$

$ChildError(p) \equiv (\exists q \in Neig_p :: (S_p = q) \wedge (Par_q = p) \wedge (S_q \neq C) \wedge \neg(Visited_p \subsetneq Visited_q))$

$LockedF(p) \equiv (\exists q \in Neig_p :: (S_q \neq C))$

$LockedB(p) \equiv [\exists q \in Neig_p :: (Id_q \notin ChildVisited_p) \wedge (S_q \neq C)] \vee ChildError(p)$

$Forward(p) \equiv (S_p = C) \wedge \neg LockedF(p)$

$Backward(p) \equiv (\exists q \in Neig_p :: (S_p = q) \wedge (Par_q = p) \wedge (S_q = D)) \wedge \neg LockedB(p)$

$Clean(p) \equiv (S_p = D)$

Règles :

$C :: Clean(p) \vee Error(p)$	$\rightarrow$	$S_p := C$;	
$F :: Forward(p) \wedge (Request_p = Wait)$	$\rightarrow$	$Visited_p := \{Id_p\}$; $S_p := Next_p$; $Request_p := In$;	/* *Démarrage* */
$B :: Backward(p)$	$\rightarrow$	$Visited_p := ChildVisited_p$; $S_p := Next_p$;	
$T :: Forward(p) \wedge (Request_p = In)$	$\rightarrow$	$Request_p := Out$;	/* *Terminaison* */

un processeur q, cette phase consiste simplement à affecter C à sa variable S_q. La règle F de *r* est l'action de démarrage du protocole : elle permet d'initier un nouveau parcours en profondeur. Dans le mouvement $i \mapsto ii$, le processeur 5 se nettoie et *r* exécute sa règle F. La règle F de *r* réalise les tâches suivantes : $Visited_r$ est initialisé avec l'identité de *r*, Id_r ; ensuite S_r est initialisé en pointant le processeur 1 qui est le voisin minimal de *r* dans l'ordre local $\prec_r$; enfin, $Request_r$ reçoit In pour signifier à l'application que la requête émise a été prise en compte par le système. Par sa règle F, *r* crée le jeton et l'envoie à 1 (son voisin minimal dans l'ordre local $\prec_r$).

Dans la configuration suivante (ii), les processeurs 2 et 3 sont encore activables pour exécuter leur phase de nettoyage. Ensuite, nous pouvons remarquer que dans cette configuration, le processeur 1 (le successeur désigné de *r*) n'est pas activable : 1 "attend" pour recevoir le jeton que tous ses voisins p tels que $S_p = D$ et $Id_p \notin PredVisited_1$ (*i.e.*, $Visited_r$) exécutent leur phase de nettoyage. En effet, grâce à l'ensemble $Visited$ de son prédécesseur ($PredVisited_1 = Visited_r$), le processeur 1 détecte que des processeurs non visités par le parcours courant n'ont pas encore exécuté leur phase de nettoyage (n.b. les processeurs 2 et 3). Dans le mouvement $ii \mapsto iii$, 2 et 3 exécutent leur phase de nettoyage (règle C) et le processeur 1 devient activable pour prolonger le parcours (règle F). Dans le mouvement $iii \mapsto iv$, 1 exécute sa règle F (n.b. 1 est leur seul processeur activable dans la configuration iii) et reçoit le jeton. Par sa règle F, le processeur 1 désigne la racine comme son père dans le parcours avec Par_p, affecte $PredVisited_1 \cup \{Id_1\}$ (*i.e.*, $Visited_r \cup \{Id_1\}$) à $Visited_1$. Ainsi nous maintenons l'invariant : l'ensemble $Visited$ du processeur qui détient le jeton contient toutes les identités des processeurs visités par le parcours. Finalement, la dernière tâche exécutée par la règle F consiste pour le processeur à se choisir comme successeur son voisin minimal par $\prec_1$ parmi ceux qui ne sont pas encore visités. Cette tâche est réalisée par l'affectation $S_1 := Next_1$. Le successeur désigné de 1 étant 4, 1 envoie le jeton à 4. Dans les mouvements suivants ($iv \mapsto v$ et $v \mapsto vi$), similairement à 1, 4 reçoit le jeton et l'envoie à 5 (règle F), puis, 5 reçoit le jeton et l'envoie à 2 (règle F). Dans la configuration vi, 2 est désigné comme successeur de 5 et n'a plus de voisin non visité par le parcours courant. Par sa règle F, 2 désigne alors 5 comme son père dans le parcours ($Par_2 := 5$), affecte $PredVisited_2 \cup \{Id_2\}$ (*i.e.*, $Visited_5 \cup \{Id_2\}$) à $Visited_2$. Enfin, comme 2

Algorithm 5.4.4 Algorithme $\mathcal{DFS}1$ pour $p \neq r$

Entrées : $Neig_p$: ensemble des voisins (localement ordonné) ; Id_p : identité de p ;

Variables : $S_p \in Neig_p \cup \{C,D\}$; $Visited_p$: ensemble d'identités ; $Par_p \in Neig_p$;

Macros :

$Next_p = (q = \min_{\prec_p}\{q' \in Neig_p :: (Id_{q'} \notin Visited_p)\})$ **si** q **existe**, D **sinon** ;
$Pred_p = \{q \in Neig_p :: (S_q = p)\}$;
$PredVisited_p = Visited_q$ **si** $(\exists! q \in Neig_p :: (S_q = p))$, $\emptyset$ **sinon** ;
$ChildVisited_p = Visited_{S_p}$ **si** $(S_p \notin \{C,D\})$, $\emptyset$ **sinon** ;

Prédicats :

$NoRealParent(p) \equiv (S_p \notin \{C, D\}) \wedge \neg(\exists q \in Neig_p :: (S_q = p) \wedge (Par_p = q))$
$SetError(p) \equiv (S_p \neq C) \wedge [(Id_p \notin Visited_p) \vee (\exists q \in Neig_p :: (S_q = q) \wedge (Id_q \in Visited_p))$
$\vee (\exists q \in Neig_p :: (S_q = p) \wedge (Par_p = q) \wedge \neg(Visited_q \subsetneq Visited_p))]$
$Error(p) \equiv NoRealParent(p) \vee SetError(p)$
$ChildError(p) \equiv (\exists q \in Neig_p :: (S_p = q) \wedge (Par_q = p) \wedge (S_q \neq C) \wedge \neg(Visited_p \subsetneq Visited_q))$
$LockedF(p) \equiv (|Pred_p| \neq 1) \vee (\exists q \in Neig_p :: (Id_q \notin PredVisited_p) \wedge (S_q \neq C))$
$\vee (Id_p \in PredVisited_p)$
$LockedB(p) \equiv (|Pred_p| \neq 1) \vee (\exists q \in Neig_p :: (Id_q \notin ChildVisited_p) \wedge (S_q \neq C))$
$\vee ChildError(p)$
$Forward(p) \equiv (S_p = C) \wedge (\exists q \in Neig_p :: (S_q = p)) \wedge \neg LockedF(p)$
$Backward(p) \equiv (\exists q \in Neig_p :: (S_p = q) \wedge (Par_q = p) \wedge (S_q = D)) \wedge \neg LockedB(p)$
$Clean(p) \equiv (S_p = D) \wedge (S_{Par_p} \neq p)$

Règles :

$C :: Clean(p) \vee Error(p) \rightarrow S_p := C$;
$F :: Forward(p) \rightarrow Visited_p := PredVisited_p \cup \{Id_p\}$; $S_p := Next_p$; $Par_p := (q \in Pred_p)$;
$B :: Backward(p) \rightarrow Visited_p := ChildVisited_p$; $S_p := Next_p$;

détecte grâce à l'ensemble $Visited_2$, qui est maintenant égal à $\{r,1,2,4,5\}$, que tous ses voisins sont visités, le processeur 2 affecte D à S_2 pour signifier à son père (5) que le parcours à partir de lui est terminé. D'où le jeton est renvoyé à 5. Dans la configuration vii, le processeur 5 détecte que le parcours à partir de 2 est terminé. Donc, 5 doit à nouveau récupérer le jeton pour continuer la phase de visite via un de ses voisins non encore visités, si un tel processeur existe. 5 exécute alors la règle B (mouvement $vii \mapsto viii$) : 5 affecte $ChildVisited_5$ (*i.e.*, $Visited_2$) à $Visited_5$. Ainsi, nous maintenons l'invariant : l'ensemble $Visited$ du processeur qui détient le jeton contient toutes les identités des processeurs visités par le parcours et 5 connait maintenant quels sont ses voisins qui ont été visités par le parcours. 5 peut alors désigner un nouveau successeur ou le cas échéant affecter D à S_p comme nous l'avons vu précédemment (cf. la règle F). Comme pour la règle F, en exécutant la règle B, p récupère le jeton et le renvoie ensuite soit à son père (si tous ses voisins sont déjà visités) soit à son nouveau successeur (un voisin non encore visité). Finalement, nous pouvons remarquer que suite à l'exécution de la règle B par 5, le processeur 2 devient activable pour exécuter sa phase de nettoyage. En effet, 2 ne fait plus partie du parcours courant et il doit donc se nettoyer pour pouvoir recevoir le prochain parcours.

Par ce mécanisme, le parcours finit par terminer à la racine par l'affection $S_r := D$ lorsque tous les processeurs ont été visités (mouvement $xii \mapsto xiii$ de la figure 5.4.5). La règle C de *r* devient alors activable pour effectuer sa phase de nettoyage. Suite au nettoyage de la racine (mouvement $xiii \mapsto xiv$ de la figure 5.4.5), la terminaison devient effective par l'exécution de la règle T qui affecte simplement Out à $Request_r$ pour signifier à l'application que le parcours demandé est terminé. Finalement, suite à l'exécution de la règle T, nous pouvons remarquer que le système atteint de nouveau une configuration initiale normale où la racine est en attente de requête.

Phase de correction. À partir du fonctionnement normal de $\mathcal{DFS}1$, nous pouvons remarquer que lorsqu'un processeur p participe à un parcours initié par la racine, il satisfait certaines propriétés. Ses propriétés vont nous permettre de détecter certains *parcours anormaux*, *i.e.*, des parcours non-initiés

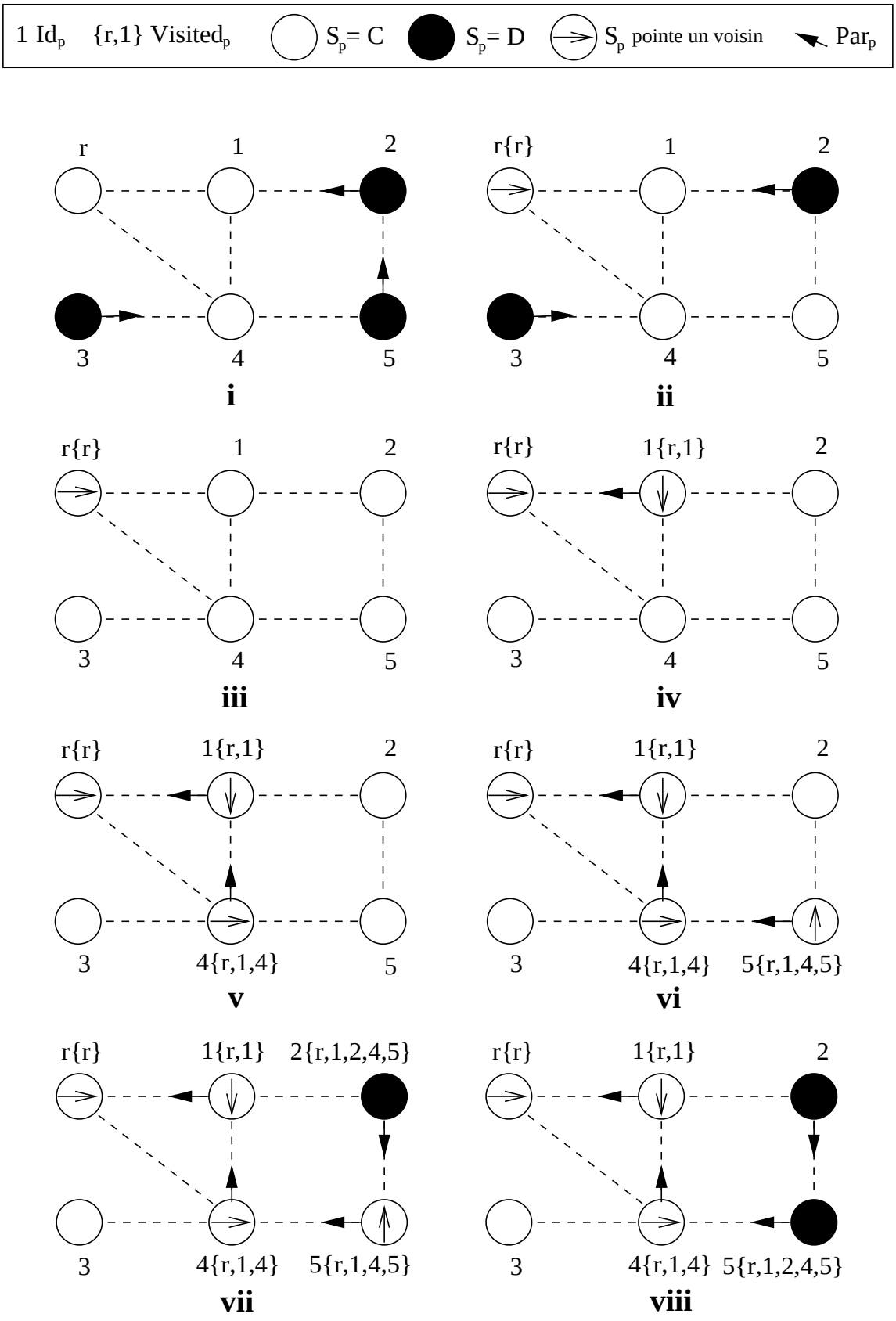


FIG. 5.4.4 – Un exemple d'un parcours effectué par le protocole $\mathcal{DFS}1$.

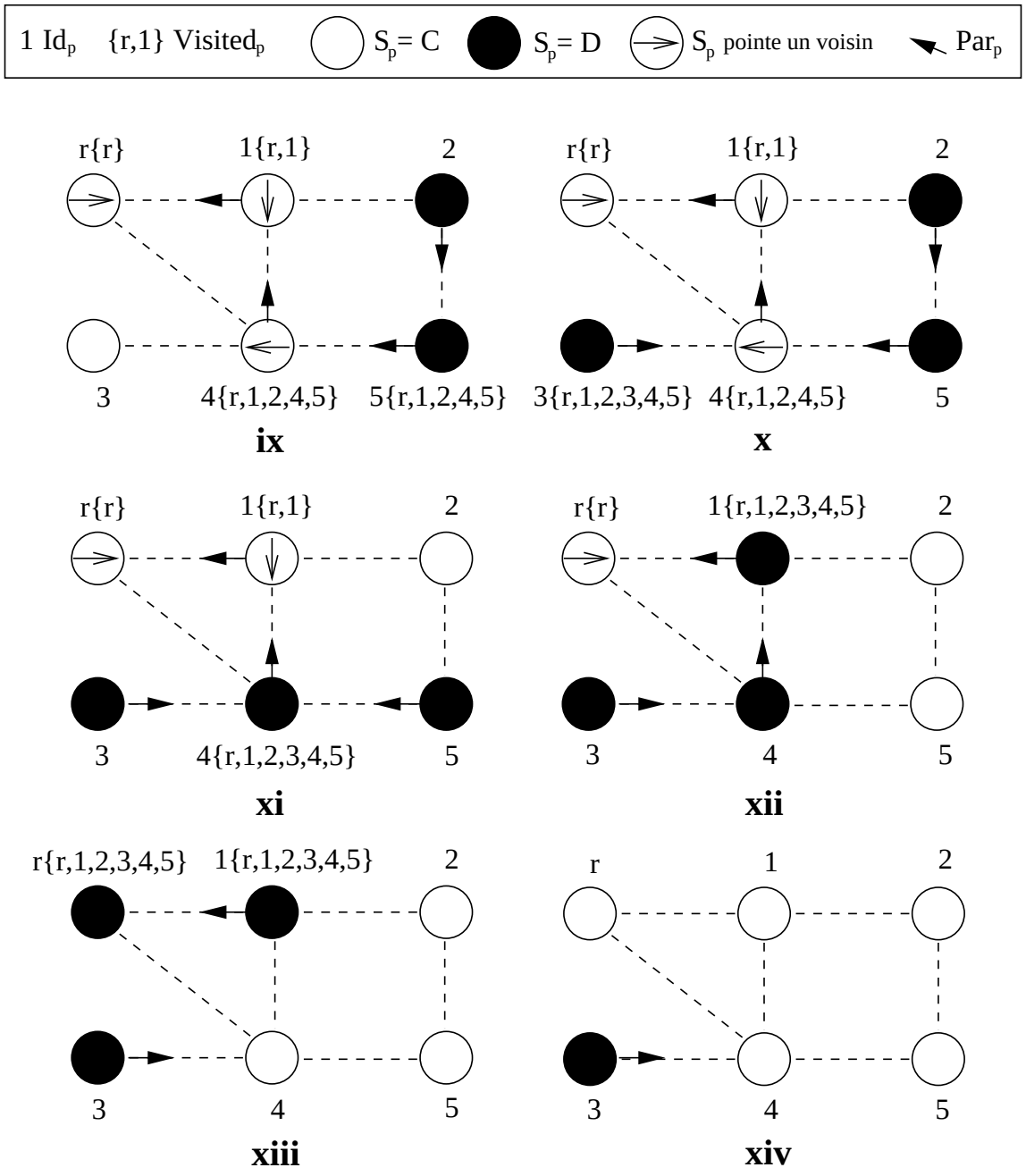


FIG. 5.4.5 – Un exemple d'un parcours effectué par le protocole $\mathcal{DFS}1$ (suite).

par la racine et de les éliminer afin de préserver le système de tout interblocage. Premièrement, pour tout processeur $p \neq r$, si p est dans une phase de visite initiée par la racine, tant qu'il n'a pas terminé cette phase de visite, il a un prédécesseur q et Par_p pointe vers q, *i.e.*, p satisfait : ($S_p \notin \{C,D\}$) $\Rightarrow$ ($\exists q \in Neig_p :: S_q = p \wedge Par_p = q$). Le prédicat $NoRealParent(p)$ permet de déterminer si cette propriété n'est pas satisfaite par p. Ensuite, dans un fonctionnement normal, chaque processeur maintient des propriétés basées sur son ensemble $Visited$ et celui de ces prédécesseurs (si de tels processeurs existent). En fait, dans chaque configuration, $\forall p \in V$, p doit respecter les conditions suivantes :

1. $(S_p \neq C) \Rightarrow (Id_p \in Visited_p)$ car p inclut son identité dans $Visited_p$ (cf. la règle F) quand il reçoit un nouveau jeton.
2. $(S_p \in Neig_p) \Rightarrow (Id_{S_p} \notin Visited_p)$. En effet, lorsque p reçoit un nouveau jeton (règle F), il recopie d'abord l'ensemble $Visited$ de son prédécesseur dans son ensemble $Visited_p$ et, ensuite, il se désigne un successeur (S_p) parmi ses voisins n'ayant pas leur identité dans $Visited_p$ (si de tels processeurs existent). Par la suite, $Visited_p$ ne sera modifié que lorsque S_p renverra le jeton à p. Par la règle B, p mettra alors son ensemble $Visited_p$ à jour en fonction des visites effectuées à partir de S_p et affectera soit D soit q tel que $q \in Neig_p \wedge Id_q \notin Visited_p$ à S_p. Donc, dans tous les cas, à partir d'une configuration initiale normale, p vérifie toujours $(S_p \in Neig_p) \Rightarrow (Id_{S_p} \notin Visited_p)$.
3. $((p \neq r) \wedge (S_p \neq C) \wedge (\exists q \in Neig_p :: (S_q = p) \wedge (Par_p = q))) \Rightarrow (Visited_q \subsetneq Visited_p)$ car, pour les mêmes raisons que celle évoquées dans le point 2, tant que $p \neq r$ exécute sa phase de visite, $Visited_p$ doit strictement inclure l'ensemble $Visited$ de son père.

Si l'une de ces conditions n'est pas satisfaite par p, alors p satisfait $SetError(p)$. Donc, dans le protocole $\mathcal{DFS}1$, p détecte s'il est dans un état "anormal", *i.e.*, $(((p \neq r) \wedge NoRealParent(p)) \vee SetError(p))$ avec le prédicat $Error(p)$. Dans la suite, nous appellerons *racine anormale* tout processeur p vérifiant $Error(p)$ (n.b. dans certains cas, r est une racine anormale, cf. le prédicat $Error(r)$ dans l'algorithme 5.4.3). Si p est une racine anormale, alors il doit se corriger ainsi que tous les processeurs visités à partir de lui : le *parcours anormal* enraciné en p. Pour cela, nous corrigeons simplement p en affectant C à S_p via la règle C (n.b., la règle C est la règle la plus prioritaire). Donc, si, avant que p exécute sa règle C, son successeur q satisfait ($S_p = q \wedge Par_q = p \wedge S_q \notin \{C, D\} \wedge \neg Error(q)$), alors après l'exécution de la règle C par p, q devient racine anormale à son tour (à la place de p). Ces corrections vont donc se propager en suivant le parcours issu de p jusqu'à ce que ce parcours ait complètement disparu. Cependant, en parallèle de ces corrections, la phase de visite issue de p (*i.e.*, le jeton) peut progresser dans le réseau par l'exécution de règles F et B. Mais, nous pouvons remarquer que, dans ce cas, l'ensemble $Visited$ du processeur qui détient le jeton grandit par les exécutions successives des règles F et B. Donc, comme la phase de visite ne se propage que via des processeurs qui ne sont pas dans cet ensemble, ce parcours anormal ne peut pas se propager infiniment longtemps. D'où, chaque phase de visite anormale (*i.e.*, chaque parcours anormal) finit par être éliminée.

Finalement, pour assurer la propriété de stabilisation instantanée, nous avons mis en oeuvre un mécanisme visant à rendre inactivable les règles F et B d'un processeur p lorsque celui-ci détecte une incohérence locale pour qu'il attende la correction de cette incohérence. Pour cela, nous avons surchargé les gardes des règles F et B avec les prédicats $LockedF$ et $LockedB$. En effet, les règles F et B gèrent la progression des jetons. Or, en observant son état local et celui de ses voisins, un processeur p peut détecter que lui ou un de ses voisins est dans un parcours anormal et, dans ce cas, se bloquer jusqu'à la correction de ce parcours : les predicats $LockedF(p)$ et $LockedB(p)$, respectivement, dans les règles F et B sont prévus à cet effet. Un processeur p se bloque dans son parcours lorsqu'il satisfait l'une des quatre conditions suivantes :

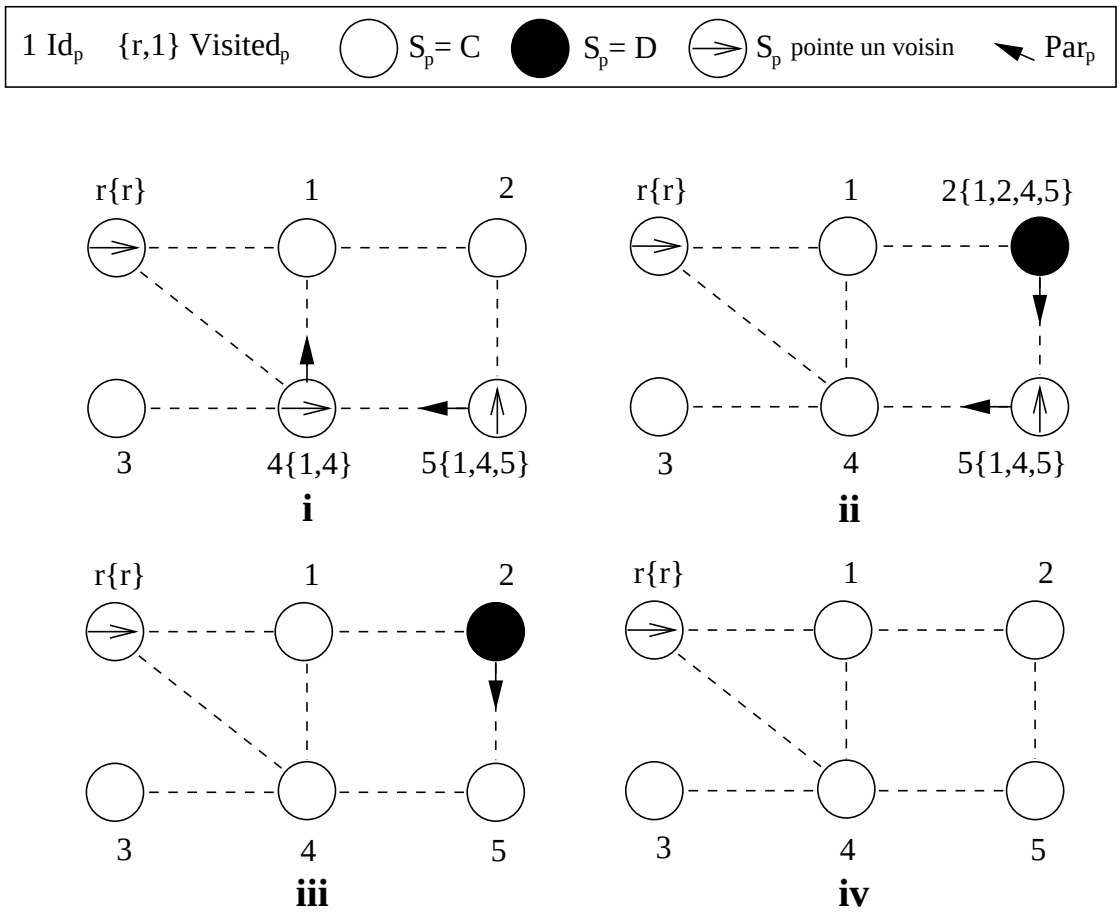


FIG. 5.4.6 – Exemple de correction d'un parcours anormal.

1. p a plusieurs prédécesseurs. Dans ce cas, p ou certains de ses prédécesseurs sont dans des parcours anormaux. Bloquer p permet alors de stopper la progression de ces parcours jusqu'à leur correction.
2. p a un successeur q tel que $((S_q \neq C) \wedge (Par_q = p) \wedge \neg(Visited_p \subsetneq Visited_q))$. Dans ce cas, q est une racine anormale et p attend que q se corrige pour qu'ensuite q participe à son parcours.
3. p vérifie $S_p = C$ et est désigné comme successeur par q mais Id_p est déjà dans $Visited_q$. Dans ce cas, p détecte que q est une racine anormale et p ne doit donc pas participer au parcours de q.
4. Certains voisins non visités de p ne sont pas encore nettoyés, *i.e.*, ils vérifient $S \neq C$ (cette dernière condition est aussi utilisée lors d'un fonctionnement normal). Soit certains de ces processeurs sont des processeurs d'un parcours "normal" précédent et sont lents à se nettoyer, soit, parmi ces processeurs, certains appartiennent à des parcours anormaux. Dans tout les cas, p attend le nettoyage de ces processeurs pour ensuite pouvoir effectuer le parcours dans le bon ordre (*i.e.* dans un ordre induit par la profondeur d'abord).

La figure 5.4.6 illustre notre méthode de correction des parcours anormaux[1]. Dans la configuration i de la figure, il existe une seule racine anormale : le processeur 4 et donc un seul parcours anormal : ce parcours est constitué des processeurs 4 et 5. Dans cette configuration, le processeur 4 est activable pour se corriger (règle C) et le processeur 2 a sa règle F activable pour participer au

[1]Dans cette figure et comme pour les figures précédentes, nous montrons uniquement, par souci de simplicité, les variables ayant une incidence sur l'exécution du protocole.

parcours anormal. En effet, de par son état local, 2 ne peut pas détecter que 5 appartient à un parcours anormal. Enfin, nous pouvons remarquer que le processeur 1 n'est pas activable dans cette configuration. En effet, la règle F de 1 n'est pas activable grâce au prédicat $LockedF(1)$: 1 suspecte une erreur car les identités des processeurs 2 et 4 ne sont pas dans l'ensemble $Visited_r$ de son prédécesseur (*r*). Donc, p pourra participer au parcours de *r* uniquement lorsque 4 et 2 seront corrigés. Dans le mouvement $i \mapsto ii$, 4 se corrige et 2 s'accroche au parcours anormal. Nous pouvons ensuite remarquer qu'à partir de la configuration ii (et comme énoncé précédemment) le parcours anormal ne peut plus progresser dans le réseau en visitant de nouveaux processeurs grâce à l'ensemble $Visited$ de son dernier élément. Dans les mouvements $ii \mapsto iii$ et $iii \mapsto iv$, le parcours finit de se corriger via l'exécution de règles C et dans la configuration iv le parcours normal se débloque car les voisins de 1 non-visités par le parcours normal, *i.e.*, 2 et 4, satisfont tous $S = C$.

5.4.2 Preuve de la stabilisation instantanée

Nous allons maintenant prouver que le protocole $\mathcal{DFS}1$ est un protocole instantanément stabilisant de parcours en profondeur sous l'hypothèse d'un démon distribué inéquitable. Tout d'abord, comme annoncé au début de ce chapitre, nous rappelons que nous considérerons dans les preuves que la valeur de la variable $Request_r$ est transparente pour le protocole (la gestion explicite des requêtes sera traitée dans la section 5.6) : nous ne tiendrons pas compte de cette variable lors de l'évalution et de l'exécution des règles. Par exemple, nous supposerons que la garde de la règle F de *r* est simplement le prédicat $Forward(r)$. Ensuite, la preuve de la stabilisation instantanée de notre protocole sous un démon distribué inéquitable sera faite en deux étapes (conformément au théorème 4.3.1, page 35) :

(i) Tout d'abord, nous allons prouver que $\mathcal{DFS}1$ est un protocole instantanément stabilisant de parcours en profondeur en supposant un démon distribué faiblement équitable (ce démon est moins général que le démon distribué inéquitable).

(ii) Puis, nous prouverons que chaque parcours effectué par $\mathcal{DFS}1$ ne peut s'exécuter qu'en un nombre fini de mouvements.

Par ces deux propositions, nous pourrons alors déduire que $\mathcal{DFS}1$ est un protocole instantanément stabilisant de parcours en profondeur sous l'hypothèse d'un démon distribué inéquitable. En effet, la proposition (ii) signifie qu'un démon distribué inéquitable ne peut pas empêcher le protocole $\mathcal{DFS}1$ d'exécuter des parcours. Or, la proposition (i) assure que $\mathcal{DFS}1$ vérifie ses spécifications dès le premier parcours.

Avant de présenter la preuve de la stabilisation instantanée de notre protocole, nous allons définir les objets que nous allons utiliser dans les preuves et prouver certaines de leurs caractéristiques.

Définitions et propriétés.

Définition 5.4.1 (Processeur pré-nettoyé) $\forall p \in V$, *p est dit* pré-nettoyé *si et seulement si p satisfait $[Clean(p) \vee (S_p = D \wedge Error(p))]$.*

Définition 5.4.2 (Racine anormale) $\forall p \in V$, *p est appelé* racine anormale *si et seulement si p satisfait $Error(p)$.*

Définition 5.4.3 (Processeurs accrochés) *Soit p et q deux processeurs distincts. q est dit* accroché *à p si et seulement si $(S_p = q) \wedge (Par_q = p) \wedge \neg Error(q) \wedge (S_q \neq C)$.*

Définition 5.4.4 (PAP : Partie Active d'un Parcours) *Nous appelons* PAP *(*partie active d'un parcours*) tout chemin $\mathcal{P} = p_1, ..., p_k$ de G tel que :*

1. $S_{p_1} \notin \{C,D\}$.
2. $\forall i$ *tel que* $1 \leq i \leq k-1$, p_{i+1} *est accroché à* p_i.

*Nous noterons $EI(\mathcal{P})$ l'extrémité initiale de $\mathcal{P}$ (*i.e., p_1*), $EF(\mathcal{P})$ l'extrémité finale de $\mathcal{P}$ (*i.e., p_k*) et $|\mathcal{P}|$ la longueur de $\mathcal{P}$ ($= k-1$).*

Lemme 5.4.1 *Tout PAP est un chemin élémentaire.*

Preuve. Supposons, par contradiction, qu'il existe une configuration du système γ dans laquelle il existe au moins un PAP qui n'est pas élémentaire. Cela signifie, en particulier, qu'il existe au moins un PAP c_1, ..., c_k dans γ avec $c_1 = c_k$. Par la définition 5.4.4, $\forall i \in [1...k-1]$, c_{i+1} est accroché à c_i. Donc, par la définition 5.4.3, $\forall i \in [1...k-1]$, nous avons ($S_{c_i} = c_{i+1}$) $\wedge$ ($Par_{c_{i+1}} = c_i$) $\wedge$ ($S_{c_{i+1}} \neq C$), et, comme $\neg Error(c_{i+1})$ et $\neg Error(c_{i+1}) \Rightarrow \neg SetError(c_{i+1})$, nous avons aussi, $Visited_{c_i} \subsetneq Visited_{c_{i+1}}$. Or, comme la relation d'inclusion est transitive, cela implique que $Visited_{c_1} \subsetneq Visited_{c_k}$, c'est à dire, $Visited_{c_1} \subsetneq Visited_{c_1}$, une contradiction. □

Remarque 5.4.1 *D'après le lemme 5.4.1, nous savons que tout PAP est un chemin de longueur finie. Donc, à partir de maintenant, nous allons considérer uniquement les PAP de longueur maximale.*

Le lemme suivant montre que l'ensemble $Visited$ de l'extrémité finale d'un PAP contient au moins les identités de tous les processeurs de ce PAP sauf, peut-être, l'identité de son extrémité initiale. En effet, d'une part, dans un PAP $\mathcal{P}$, seule l'extrémité initiale, $EI(\mathcal{P})$, peut vérifier le prédicat $Error$, donc, $EI(\mathcal{P})$ est le seul processeur dans $\mathcal{P}$ qui peut ne pas contenir son identité dans son ensemble $Visited$. D'autre part, comme les autres processeurs de $\mathcal{P}$ vérifient $\neg Error$, ils contiennent au moins leur identité et les identités de l'ensemble $Visited$ de leur prédécesseur dans $\mathcal{P}$. D'où, par transitivité, l'ensemble $Visited$ de l'extrémité finale de $\mathcal{P}$ contient au moins les identités de tous les processeurs de $\mathcal{P}$ sauf, peut-être, l'identité de l'extrémité initiale.

Pour prouver formellement ce lemme, nous allons utiliser une notion de "distance" entre un processeur p appartenant à un PAP et l'extrémité initiale de ce PAP. Soit $\mathcal{P}$ un PAP. La distance entre p et $EI(\mathcal{P})$ (*i.e.*, l'extrémité initiale de $\mathcal{P}$), notée $d(p,\mathcal{P})$, est définie récursivement comme suit :

– $d(p,\mathcal{P}) = 0$, **si** $p = EI(\mathcal{P})$.
– $d(p,\mathcal{P}) = d(q,\mathcal{P}) + 1$ où q est le processeur à qui p est accroché dans $\mathcal{P}$, **sinon**.

Lemme 5.4.2 *Soit $\mathcal{P}$ un PAP. $\mathcal{P}$ satisfait $Visited_{EF(\mathcal{P})} \supseteq \{Id_p :: p \in \mathcal{P} \wedge p \neq EI(\mathcal{P})\}$.*

Preuve. Soit $\mathcal{P}$ un PAP. Pour prouver ce lemme, nous allons montrer par récurrence sur $d(p, \mathcal{P})$ que $\forall p \in \mathcal{P}$, $Visited_p \supseteq \{Id_{p'} :: p' \in \mathcal{P} \wedge d(p', \mathcal{P}) \leq d(p, \mathcal{P}) \wedge p' \neq EI(\mathcal{P})\}$. En effet, si nous appliquons cette propriété à $EF(\mathcal{P})$, le lemme est trivialement vérifié.

Soit i le processeur de $\mathcal{P}$ tel que $d(i, \mathcal{P}) = 0$. Par définition, $i = EI(\mathcal{P})$ et il n'existe pas de processeur $i' \in \mathcal{P}$ tel que $d(i', \mathcal{P}) \leq d(EI(\mathcal{P}),\mathcal{P}) \wedge i' \neq EI(\mathcal{P})$. Cela signifie, en particulier, que $\{Id_{i'} :: i' \in \mathcal{P} \wedge d(i',\mathcal{P}) \leq d(i,\mathcal{P}) \wedge i' \neq EI(\mathcal{P})\} = \emptyset$ et la récurrence est trivialement vérifiée $d(i,\mathcal{P}) = 0$ ($Visited_i \supseteq \emptyset$).

Supposons maintenant que, pour $d \geq 0$, $\forall p \in \mathcal{P}$ tel que $d(p,\mathcal{P}) \leq d$, p vérifie $Visited_p \supseteq \{Id_{p'} :: p' \in \mathcal{P} \wedge d(p',\mathcal{P}) \leq d(p,\mathcal{P}) \wedge p' \neq EI(\mathcal{P})\}$.

Soit $q \in \mathcal{P}$ tel que $d(q,\mathcal{P}) = d+1$. Comme $d(q,\mathcal{P}) \geq 1$, par les définitions 5.4.3 and 5.4.4, $S_q \neq C \wedge \neg SetError(q)$ ($\neg Error(q) \Rightarrow \neg SetError(q)$). Donc, $Id_q \in Visited_q$. Par la définition 5.4.4, $\exists p \in \mathcal{P}$ tel que q est accroché à p. De plus, $q \neq r$ (comme $Par_r = \perp$, par la définition 5.4.3, r ne peut

être accroché à aucun processeur). Or, comme $d(p,\mathcal{P}) = d$, $Visited_p \supseteq \{Id_{p'} :: p' \in \mathcal{P} \wedge d(p',\mathcal{P}) \leq d \wedge p' \neq EI(\mathcal{P})\}$ par hypothèse de récurrence. De plus, comme $q \neq r \wedge S_q \neq C \wedge \neg SetError(q)$, nous avons $Visited_p \subsetneq Visited_q$. D'où, $Visited_q \supseteq (\{Id_{p'} :: p' \in \mathcal{P} \wedge d(p',\mathcal{P}) \leq d \wedge p' \neq EI(\mathcal{P})\} \cup \{Id_q\})$, c'est à dire, $Visited_q \supseteq \{Id_{q'} :: q' \in \mathcal{P} \wedge d(q',\mathcal{P}) \leq d(q,\mathcal{P}) \wedge q' \neq EI(\mathcal{P})\}$ et la récurrence est vérifiée pour les processeurs p tel que $p \in \mathcal{P}$ et $d(p,\mathcal{P}) \leq d+1$. □

Nous allons maintenant différencier deux types de PAP : les PAPs dit *normaux* et les PAPs dit *anormaux*.

Définition 5.4.5 *[PAPs normaux et anormaux] Nous appelons* PAP anormal *tout PAP $\mathcal{P}$ tel que $Error(EI(\mathcal{P}))$. Nous appelons* PAP normal*, tout PAP qui n'est pas anormal.*

Remarque 5.4.2 *Soit $\mathcal{P}$ un PAP. Par définition, si $\mathcal{P}$ est normal, alors $EI(\mathcal{P}) =$ r. En revanche, $EI(\mathcal{P}) =$ r n'implique pas forcément que $\mathcal{P}$ soit normal, en effet,* r *peut vérifier $Error(r)$.*

Le lemme suivant montre que l'ensemble $Visited$ de l'extrémité finale d'un PAP normal contient au moins les identités de tous les processeurs du PAP.

Lemme 5.4.3 *Tout PAP normal $\mathcal{P}$ vérifie $Visited_{EF(\mathcal{P})} \supseteq \{Id_p :: p \in \mathcal{P}\}$.*

Preuve. Soit $\mathcal{P}$ un PAP normal. D'après la remarque 5.4.2, $EI(\mathcal{P}) = r$ et comme $S_r \neq C \wedge \neg Error(r)$, nous avons aussi $Id_r \in Visited_r$. Donc, $Visited_{EI(\mathcal{P})} \supseteq \{Id_{EI(\mathcal{P})}\}$. Ainsi, le lemme peut être vérifié par récurrence sur $d(x,\mathcal{P})$ comme dans la preuve du lemme 5.4.2. □

Nous introduisons maintenant la notion de *futur* d'un PAP. Cette notion nous permettra d'étudier l'évolution d'un PAP au cours de l'exécution. En particulier, le *futur immédiat* d'un PAP $\mathcal{P}$ représente la transformation subie par $\mathcal{P}$ après un mouvement. Un PAP peut disparaître après un mouvement. Par convention, nous exprimerons par $Dead_{\mathcal{P}}$ le fait que le PAP $\mathcal{P}$ ait disparu après un mouvement.

Définition 5.4.6 (Futur immédiat d'un PAP) *Soit $\gamma_i \mapsto \gamma_{i+1}$ un mouvement. Soit $\mathcal{P}$ un PAP existant dans γ_i. Nous appelons $\mathcal{F}(\mathcal{P})$* le futur immédiat *de $\mathcal{P}$ dans γ_{i+1} et $\mathcal{F}(\mathcal{P})$ est défini comme suit :*

1. **Si** *il existe un PAP $\mathcal{P}'$ dans γ_{i+1} qui vérifie l'une des deux conditions suivantes :*
   (a) *$\mathcal{P} \cap \mathcal{P}' \neq \emptyset$, ou*
   (b) *$S_{EF(\mathcal{P})}$ dans γ_i est égal à $EI(\mathcal{P}')$ dans γ_{i+1} et $EI(\mathcal{P}')$ a exécuté la règle F dans $\gamma_i \mapsto \gamma_{i+1}$*

   alors *$\mathcal{F}(\mathcal{P}) = \mathcal{P}'$,*
2. **Sinon**, *$\mathcal{F}(\mathcal{P}) = Dead_{\mathcal{P}}$. Dans ce cas, $\mathcal{P}$ sera dit* mort.

Par convention, nous supposerons que $\mathcal{F}(Dead_{\mathcal{P}}) = Dead_{\mathcal{P}}$.

La figure 5.4.7 illustre la notion de futur immédiat. Considérons, tout d'abord, les configurations $(a).i$ et $(a).ii$. La configuration $(a).i$ contient un seul PAP : $\mathcal{P} = r$, 1, 2. Ensuite, le processeur 3 exécute sa règle F dans le mouvement $(a).i \mapsto (a).ii$: 3 s'accroche à $\mathcal{P}$. Donc, le mouvement $(a).i \mapsto (a).ii$ illustre le cas 1.(a) de la définition 5.4.6 : dans la configuration $(a).ii$, $\mathcal{F}(\mathcal{P}) = r$, 1, 2, 3. La configuration $(b).i$ contient aussi un seul PAP : $\mathcal{P}' = 1$. De plus, les règles C du processeur 1 et F du processeur 2 sont activables dans $(b).i$. Donc, si ces deux règles sont exécutés dans le même mouvement (*i.e.*, 1 quitte $\mathcal{P}'$ et 2 s'accroche à $\mathcal{P}'$), nous obtenons la configuration $(b).ii$ qui illustre le cas 1.(b) de la définition 5.4.6 : dans la configuration $(b).ii$, $\mathcal{F}(\mathcal{P}') = 2$. Enfin, notez que si seule la règle C du processeur 1 avait été exécutée lors de $(b).i \mapsto (b).ii$, $\mathcal{P}'$ aurait disparu et, dans ce cas, $\mathcal{F}(\mathcal{P}')$ aurait été égal à $Dead_{\mathcal{P}'}$ dans $(b).ii$, *i.e.* le cas 2 de la définition 5.4.6.

Avec la définition suivante, nous généralisons la notion de futur immédiat.

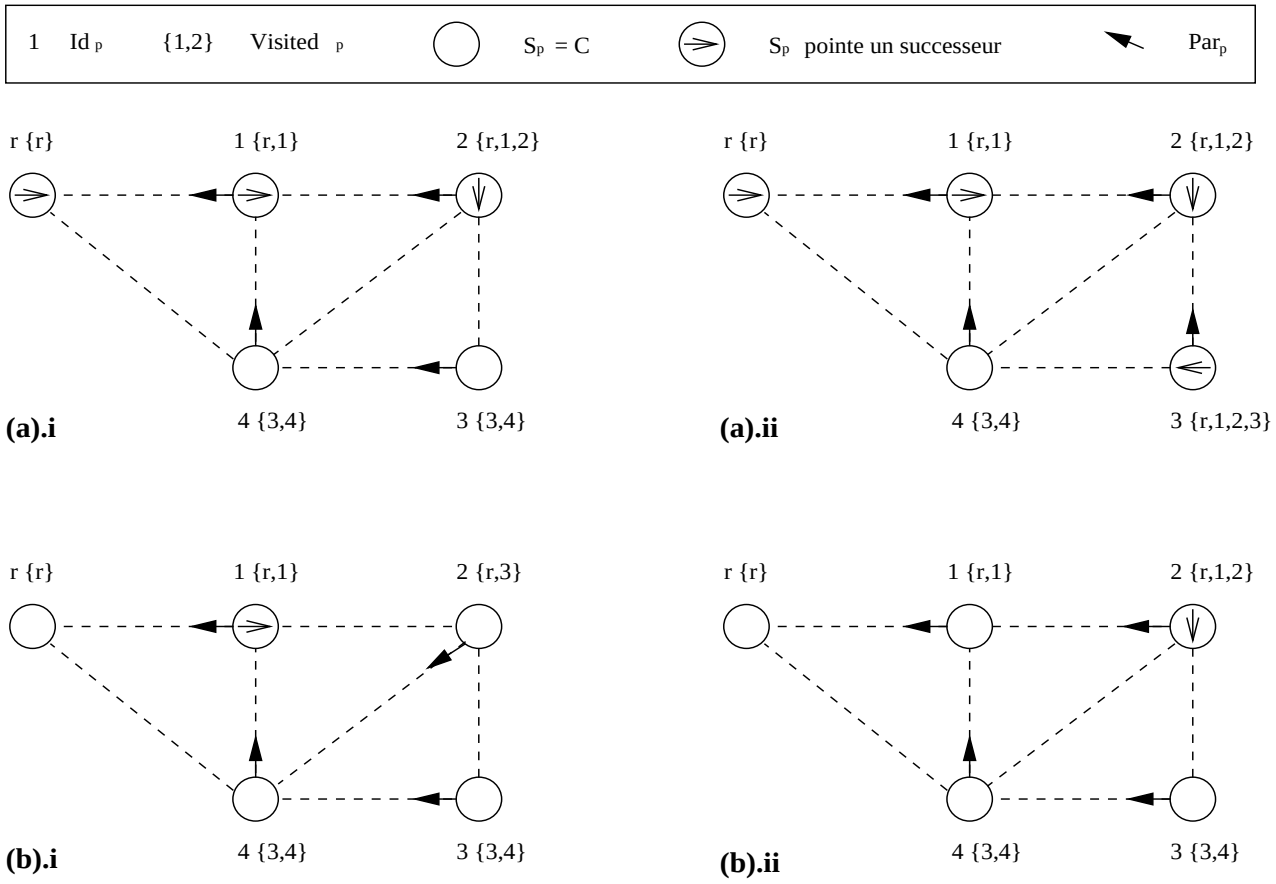


FIG. 5.4.7 – Exemples de futurs immédiats.

Définition 5.4.7 (Futur d'un PAP) *Soit $e \in \mathcal{E}$, $\gamma_i \in e$ et $\mathcal{P}$ un PAP dans γ_i. Nous définissons $\mathcal{F}^k(\mathcal{P})$, $\forall k \in \mathbb{N}$, le futur de $\mathcal{P}$ après k mouvements dans e à partir de la configuration γ_i comme suit :*

1. $\mathcal{F}^0(\mathcal{P}) = \mathcal{P}$,
2. $\mathcal{F}^1(\mathcal{P}) = \mathcal{F}(\mathcal{P})$ *(le futur immédiat de $\mathcal{P}$),*
3. $\mathcal{F}^k(\mathcal{P}) = \mathcal{F}^{k-1}(\mathcal{F}(\mathcal{P}))$ *(le futur de $\mathcal{P}$ après k mouvements),* **si** $k > 1$.

Lemme 5.4.4 *Soit $\mathcal{P}$ un PAP. Tant que $\mathcal{F}^k(\mathcal{P}) \neq Dead_{\mathcal{P}}$ ($k \in \mathbb{N}$), $Visited_{EF(\mathcal{F}^k(\mathcal{P}))}$ contient exactement $Visited_{EF(\mathcal{P})}$ union l'ensemble des identités des processeurs qui se sont accrochés à $\mathcal{P}$ ou à ses futurs jusqu'à $\mathcal{F}^k(\mathcal{P})$).*

Preuve. Soit $\mathcal{P}$ un PAP dans la configuration γ_0. Nous allons prouver ce lemme par récurrence sur k, le nombre de mouvements exécutés depuis γ_0.

Pour $k = 0$, la récurrence est triviallement vérifiée ($\mathcal{F}^0(\mathcal{P}) = \mathcal{P}$).

Supposons qu'après k mouvements ($k \geq 0$), $Visited_{EF(\mathcal{F}^k(\mathcal{P}))}$ contient exactement $Visited_{EF(\mathcal{P})}$ union l'ensemble des identités des processeurs qui se sont accrochés à $\mathcal{P}$ ou à ses futurs jusqu'à $\mathcal{F}^k(\mathcal{P})$).

Étudions maintement la valeur de $Visited_{EF(\mathcal{F}^{k+1}(\mathcal{P}))}$ après le mouvement $\gamma_k \mapsto \gamma_{k+1}$.

– **Si** $Visited_{EF(\mathcal{F}^{k+1}(\mathcal{P}))} = Visited_{EF(\mathcal{F}^k(\mathcal{P}))}$ **alors**, d'après les algorithmes 5.4.3 et 5.4.4, l'extrémité finale de $\mathcal{F}^{k+1}(\mathcal{P})$ est soit la même que celle de $\mathcal{F}^k(\mathcal{P})$ (*i.e.*, $EF(F^k(\mathcal{P}))$) soit le prédécesseur p de $EF(F^k(\mathcal{P}))$ dans γ_k (dans ce cas, p a recopié $Visited_{EF(\mathcal{F}^k(\mathcal{P}))}$ dans $Visited_p$ en exécutant sa règle B). En particulier, cela signifie qu'aucun nouveau processeur ne

s'est accroché à $\mathcal{F}^k(\mathcal{P})$ durant $\gamma_k \mapsto \gamma_{k+1}$ et, par hypothèse de récurrence, la récurrence est vérifiée pour $k+1$.

– **Sinon** ($Visited_{EF(\mathcal{F}^{k+1}(\mathcal{P}))} \neq Visited_{EF(\mathcal{F}^k(\mathcal{P}))}$). Dans ce cas, d'après les algorithmes 5.4.3 et 5.4.4, l'extrémité finale de $\mathcal{F}^k(\mathcal{P})$, *i.e.*, $EF(\mathcal{F}^{k+1}(\mathcal{P}))$ correspond à l'unique processeur p qui s'est accroché à $\mathcal{F}^k(\mathcal{P})$ en exécutant sa règle F dans $\gamma_k \mapsto \gamma_{k+1}$. Or, par l'exécution de cette règle, $Visited_p := Visited_{EF(\mathcal{F}^k(\mathcal{P}))} \cup \{Id_p\}$ et, par hypothèse de récurrence, la récurrence est vérifiée pour $k+1$.

D'où, le lemme est vérifié. □

Lemme 5.4.5 *Soit $\mathcal{P}$ un PAP. $\forall\, p \in V$ tel que $Id_p \in Visited_{FE(P)}$, p ne peut pas s'accrocher à $\mathcal{P}$.*

Preuve. Soit $\gamma_i \mapsto \gamma_{i+1}$ un mouvement. Soit $\mathcal{P}$ un PAP existant dans γ_i. Supposons, par contradiction, qu'un processeur p s'accroche à $\mathcal{P}$ tandis que $Id_p \in Visited_{FE(P)}$. Tout d'abord, par définition 5.4.3, $p \neq r$ (en effet, Par_r est égal à la constante $\perp$ donc *r* ne s'accroche jamais à aucun PAP). Ensuite, d'après les algorithmes 5.4.4, p s'accroche à $\mathcal{P}$ dans $\gamma_i \mapsto \gamma_{i+1}$ en exécutant sa règle F et F est exécutée dans $\gamma_i \mapsto \gamma_{i+1}$ seulement si $\neg LockedF(p)$. Or, $\neg LockedF(p) \Rightarrow (Id_p \notin PredVisited_p) \Rightarrow (Id_p \notin Visited_{FE(P)})$, contradiction. □

Nous déduisons le lemme suivant d'après les lemmes 5.4.4 et 5.4.5.

Lemme 5.4.6 *Soit $\mathcal{P}$ un PAP. Si $p \in V$ s'accroche à $\mathcal{P}$, alors p ne peut plus s'accrocher à $\mathcal{F}^k(\mathcal{P})$, $\forall k \in \mathbb{N}^+$.*

Dans la suite, nous allons étudier l'évolution des PAPs au cours du temps. Donc, beaucoup de résultats vont concerner les PAPs, $\mathcal{P}$, et leurs futurs, $\mathcal{F}(\mathcal{P})^k$, $\forall k \in \mathbb{N}$. Par souci de simplicité, à partir de maintenant, lorsqu'il n'y aura pas d'ambiguïté, nous parlerons simplement de $\mathcal{P}$ pour faire référence à $\mathcal{P}$ et ses futurs, $\mathcal{F}(\mathcal{P})^k$. Par exemple, nous pourrons reformuler le lemme 5.4.5 comme suit : un processeur ne peut s'accrocher qu'une seule fois à un PAP donné.

Preuve de la stabilisation instantanée en supposant un démon distribué faiblement équitable. Nous supposons maintenant que le démon distribué est faiblement équitable. Sous cette hypothèse, le nombre de mouvements par ronde est fini. Donc, comme nous avons défini le futur d'un PAP en terme de mouvements, nous pouvons aussi l'évaluer en terme de rondes. Soit $e \in \mathcal{E}$. Soit $\mathcal{P}$ un PAP existant dans la configuration γ_i telle que $\gamma_i \in e$. Nous notons $\mathcal{F}_R^K(\mathcal{P})$ le futur de $\mathcal{P}$ après K rondes dans e à partir de γ_i.

La première étape de notre preuve consiste à montrer que le système ne contient plus de PAP anormal en au plus n rondes, *i.e.*, tout PAP anormal $\mathcal{P}$ de la configuration initiale satisfait $\mathcal{F}_R^n(\mathcal{P}) = Dead_{\mathcal{P}}$.

La remarque technique suivante est utilisée dans la preuve du lemme 5.4.7.

Remarque 5.4.3 *Dans un mouvement $\gamma_i \mapsto \gamma_{i+1}$, un processeur p peut affecter $q \in Neig_p$ à S_p seulement si $S_q = C$ dans γ_i (cf. les prédicats $LockedF(p)$ et $LockedB(p)$).*

Lemme 5.4.7 *Lorsque la règle C d'un processeur p est activable, elle reste activable jusqu'à ce que p l'exécute.*

Preuve. Pour prouver ce lemme, nous allons montrer que si la règle C d'un processeur p est activable dans la configuration γ_i et que p n'exécute pas cette règle dans $\gamma_i \mapsto \gamma_{i+1}$, alors elle reste activable dans γ_{i+1}.

Tout d'abord, d'après les algorithmes 5.4.3 et 5.4.4, la règle C de p est activable dans γ_i si et seulement si $Clean(p) \vee ((p \neq r) \wedge NoRealParent(p)) \vee SetError(p)$, avec en particulier, $S_p \neq C$ dans γ_i. De plus, C est la règle la plus prioritaire de p donc cela signifie que p n'exécute pas de règles dans $\gamma_i \mapsto \gamma_{i+1}$. Nous divisons maintenant notre étude en trois cas :

1. Supposons que p satisfait $Clean(p)$ dans γ_i.
   - Si $p = r$, alors $Clean(p)$ est défini uniquement avec les variables de r ($Clean(p) \equiv (S_p = D)$). Donc, comme r n'exécute pas de règle dans $\gamma_i \mapsto \gamma_{i+1}$, r satisfait encore $Clean(r)$ et sa règle C est encore activable dans γ_{i+1}.
   - Si $p \neq r$, alors, d'après $Clean(p)$ (pour $p \neq r$), $S_{Par_p} \neq p \wedge S_p \neq C$ dans γ_i. Comme p n'exécute pas de règle dans $\gamma_i \mapsto \gamma_{i+1}$, nous avons (par la remarque 5.4.3) $S_{Par_p} \neq p \wedge S_p \neq C$ dans γ_{i+1}. Donc, $Clean(p)$ reste vrai et la règle C de p reste activable dans γ_{i+1}.
2. Supposons que p satisfait $((p \neq r) \wedge NoRealParent(p))$ dans γ_i. Ce cas est similaire au cas précédent pour $p \neq r$. Donc, p reste activable dans γ_{i+1}.
3. Supposons que p satisfait $SetError(p)$ dans γ_i.
   - Supposons que $(Id_p \notin Visited_p) \vee (S_p \notin \{C, D\} \wedge Id_{S_p} \in Visited_p)$. Dans ce cas, $SetError(p)$ est vrai à cause des variables de p uniquement. Or, comme p n'exécute pas de règle dans $\gamma_i \mapsto \gamma_{i+1}$, $SetError(p)$ reste vrai et la règle C de p reste activable dans γ_{i+1}.
   - Supposons que $((p \neq r) \wedge (\exists q \in Neig_p :: (S_q = p) \wedge (Par_p = q) \wedge \neg(Visited_q \subsetneq Visited_p)))$ dans γ_i. Nous pouvons alors remarquer que $SetError(p)$ dépend de l'état des processeurs p et q. Ensuite, comme p n'exécute pas de règle dans $\gamma_i \mapsto \gamma_{i+1}$, si q n'exécute pas de règle dans $\gamma_i \mapsto \gamma_{i+1}$, alors la règle C de p reste activable dans γ_{i+1}. Supposons alors que q exécute une règle dans $\gamma_i \mapsto \gamma_{i+1}$. Le processeur q peut uniquement exécuter sa règle C dans $\gamma_i \mapsto \gamma_{i+1}$ (si $Error(q)$ est satisfait). En effet, la règle B de q n'est pas activable à cause de $ChildError(q)$ et la règle F de q n'est pas activable car $S_q \neq C$. Maintenant, si q exécute C dans $\gamma_i \mapsto \gamma_{i+1}$, alors $(Clean(p) \vee NoRealParent(p))$ est vrai dans γ_{i+1} et la règle C de p reste activable dans γ_{i+1}.

Dans tous les cas, si p n'exécute pas C dans $\gamma_i \mapsto \gamma_{i+1}$, alors C reste activable pour p dans γ_{i+1} et le lemme est vérifié. □

Théorème 5.4.1 *Le système ne contient plus de PAP anormal en au plus n rondes.*

Preuve. Soit $\mathcal{P}$ un PAP anormal. Par la définition 5.4.5, nous avons $Error(EI(\mathcal{P}))$ et la règle C de $EI(\mathcal{P})$ est continûment activable (Lemme 5.4.7). Or, comme le démon est faiblement équitable, $EI(\mathcal{P})$ exécute sa règle C en au plus une ronde. Ensuite, si $\mathcal{F}_R^1(\mathcal{P}) \neq Dead_{\mathcal{P}}$, alors, par la définition 5.4.5, la règle C de $EI(F_R^1(\mathcal{P}))$ est aussi continûment activable. Ainsi, tant qu'il existe des PAPs anormaux, au moins un processeur se décroche de chaque PAP anormal à chaque ronde. Or, pour chaque PAP anormal $\mathcal{P}$, seuls peuvent s'accrocher à $\mathcal{P}$ durant l'exécution, une fois et une seule, les processeurs p tel que $p \neq r$ (car la racine ne peut s'accrocher à aucun PAP anormal) et $Id_p \notin Visited_{EF(\mathcal{P})}$ (lemmes 5.4.4 et 5.4.5). Ensuite, par le lemme 5.4.2, le nombre de ces processeurs est bornés par $(n-1)$ - ($|\mathcal{P}|$ - 1), *i.e.*, $(n - |\mathcal{P}|)$ où $|\mathcal{P}|$ est la longueur de $\mathcal{P}$ dans la configuration initiale. Donc, dans le pire des cas, n rondes sont nécessaires (*i.e.*, $|\mathcal{P}|$ + $(n - |\mathcal{P}|)$) pour décrocher les processeurs initialement dans $\mathcal{P}$ et ceux qui se sont accrochés à $\mathcal{P}$ durant l'exécution. D'où, pour chaque PAP anormal $\mathcal{P}$ de la configuration initiale, nous avons $\mathcal{F}(\mathcal{P})_R^n(\mathcal{P}) = Dead_{\mathcal{P}}$, *i.e.*, le système ne contient plus de PAP anormal en au plus n rondes. □

Les lemmes et théorèmes suivants vont nous permettre de prouver, qu'à partir de n'importe quelle configuration initiale, la racine initie, via sa règle F, un parcours en un nombre fini de rondes (théorème 5.4.3).

Lemme 5.4.8 *Soit $\mathcal{P}$ un PAP normal. $\mathcal{P}$ est* mort *après l'exécution d'au plus $2n-2$ actions d'accrochage et de décrochage.*

Preuve. Soit $e \in \mathcal{E}$ une exécution. Soit γ_i une configuration de e. Supposons qu'il existe un PAP normal $\mathcal{P}$ dans γ_i. D'après l'algorithme, deux types de règle peuvent être exécutées sur $\mathcal{P}$: F et B. La règle F permet à un processeur de s'accrocher à $\mathcal{P}$. Tandis que l'exécution de la règle B provoque le décrochage de l'extrémité finale de $\mathcal{P}$ ($EF(\mathcal{P})$) : en effet, à tout moment, seul le prédécesseur p de l'extrémité finale d'un PAP peut exécuter sa règle B et, par cette règle, soit il exécute $S_p := D$ soit il désigne un nouveau successeur.

Par les lemmes 5.4.5 et 5.4.6, seuls les processeurs p tels que $Id_p \notin Visited_{FE(P)}$ dans γ_i peuvent s'accrocher à $\mathcal{P}$ durant l'exécution (via la règle F) et cela au plus une fois. Donc, au plus $n - |\mathcal{P}|$ processeurs peuvent s'accrocher à $\mathcal{P}$ où $|\mathcal{P}|$ est la longueur de $\mathcal{P}$ dans γ_i (lemme 5.4.3).

Ensuite, dans le pire des cas, après que $n-2$ processeurs se sont décrochés de $\mathcal{P}$ via l'exécution de la règle B par leurs prédécesseurs, le système est dans une configuration où $\mathcal{P}$ vérifie $|\mathcal{P}| = 2$ et une seule règle peut être exécutée sur $\mathcal{P}$: le père de $EF(\mathcal{P})$, *i.e.*, $EI(\mathcal{P})$, peut exécuter sa règle B. Or, par le lemme 5.4.4, dans cette configuration, $Visited_{EF(\mathcal{P})} = \{Id_q :: q \in V\}$. Donc, par l'exécution de la règle B, $EI(\mathcal{P})$ affecte D à $S_{EI(\mathcal{P})}$ (cf. $Next_{EI(\mathcal{P})}$) et $\mathcal{P}$ disparaît ($\mathcal{F}(\mathcal{P}) = Dead_{\mathcal{P}}$).

Ainsi, dans le pire des cas, $\mathcal{P}$ est mort après avoir subi l'exécution de $n - |\mathcal{P}| + (n-2) + 1$ règles où $|\mathcal{P}|$ est la longueur de $\mathcal{P}$ dans γ_i. Ce nombre est maximal avec $|\mathcal{P}| = 1$ dans γ_i, *i.e.*, $2n-2$ règles. □

Lemme 5.4.9 *Soit $\mathcal{P}$ un PAP normal. Si il n'existe pas de PAP anormal dans le système, alors $\mathcal{F}_R^{2n-1}(\mathcal{P}) = Dead_{\mathcal{P}}$.*

Preuve. Soit $e \in \mathcal{E}$ une exécution. Soit $\gamma_i \in e$ une configuration de e dans laquelle il existe un unique PAP : le PAP normal $\mathcal{P}$.

Tout d'abord, à partir de γ_i, pour tout processeur p, si $p \notin \mathcal{P}$, alors $S_p = C$ ou p est pré-nettoyé. Ensuite, tous les processeurs pré-nettoyés dans γ_i ont leur règle C continûment activable (Lemme 5.4.7) et comme le démon est faiblement équitable, ils exécutent leur règle C en au plus une ronde. Donc, après au plus une ronde à partir de γ_i, tout processeur pré-nettoyé existant dans le système a été généré pendant l'exécution par le PAP normal $\mathcal{P}$. D'où, $\forall k \geq 1$, si $\mathcal{F}_R^k(\mathcal{P}) \neq Dead_{\mathcal{P}}$, alors les identités de tous les processeurs pré-nettoyés sont dans l'ensemble $Visited_{EF(\mathcal{F}_R^k(\mathcal{P}))}$ (cf. lemme 5.4.4). Donc, $\forall k \geq 1$, si $\mathcal{F}_R^k(\mathcal{P}) \neq Dead_{\mathcal{P}}$, deux cas sont possibles au début de la $(k+1)$ème ronde à partir de γ_i :

- $S_{EF(\mathcal{F}_R^k(\mathcal{P}))} = D$. Soit p le père de $EF(\mathcal{F}_R^k(\mathcal{P}))$. Tout d'abord, puisqu'il n'existe plus de PAP anormal, nous avons $|Pred_p| = 1$. Ensuite, par la discussion précédente et le lemme 5.4.3, nous savons que les identités de tous les processeurs pré-nettoyés ainsi que les identités de tous les processeurs de $\mathcal{F}_R^k(\mathcal{P})$ sont dans l'ensemble $Visited_{EF(\mathcal{F}_R^k(\mathcal{P}))}$. De plus, $ChildVisited_p = Visited_{EF(\mathcal{F}_R^k(\mathcal{P}))}$. Donc, $\forall q \in Neig_p$ tel que $Id_q \notin ChildVisited_p$, nous avons $S_q = C$. D'où, p satisfait $Backward(p) \wedge \neg LockedB(p)$ et sa règle B est continûment activable car, excepté p, seul les processeurs pré-nettoyés peuvent exécuter une règle : la règle C.
- $S_{EF(\mathcal{F}_R^k(\mathcal{P}))} \neq D$. Soit p le processeur tel que $p = S_{EF(\mathcal{F}_R^k(\mathcal{P}))}$. Comme précédemment, nous avons $|Pred_p| = 1$ et $\forall q \in Neig_p$ tel que $Id_q \notin PredVisited_p$, q satisfait $S_q = C$. D'où, p satisfait $Forward(p) \wedge \neg LockedF(p)$ et la règle F de p est continûment activable.

Donc, après une ronde à partir de γ_i, au début de chaque nouvelle ronde, exactement une règle permettant de faire progresser le PAP normal est continûment activable jusqu'à ce qu'il soit mort. Dans le pire des cas, une règle est bien exécutée sur le PAP normal par ronde et, par le lemme 5.4.8, nous avons $\mathcal{F}_R^{1+(2n-2)}(\mathcal{P}) = Dead_{\mathcal{P}}$, *i.e.*, $\mathcal{F}_R^{2n-1}(\mathcal{P}) = Dead_{\mathcal{P}}$. □

Par le théorème 5.4.1 et le lemme 5.4.9, nous obtenons :

Théorème 5.4.2 *Tout PAP normal $\mathcal{P}$ satisfait $\mathcal{F}_R^{3n-1}(\mathcal{P}) = Dead_{\mathcal{P}}$.*

Théorème 5.4.3 *À partir de n'importe quelle configuration initiale,* r *exécute sa règle F après au plus $3n$ rondes.*

Preuve. À partir de n'importe quelle configuration initiale, le système atteint en au plus $3n - 1$ rondes une configuration γ_i telle que $\forall p \in V$, $S_p \in \{C,D\}$ (théorèmes 5.4.1 et 5.4.2). Dans γ_i, $\forall p \in V$ tel que $S_p = D$ nous avons $S_{Par_p} \neq p$. Donc, chaque p est pré-nettoyé et a sa règle C continûment activable (lemme 5.4.7). Or, comme le démon est faiblement équitable, dans le pire des cas, après une ronde, nous avons $\forall p \in V$, $S_p = C$ et, la règle F de *r* est activable dans le système. D'où, à partir de n'importe quelle configuration initiale, *r* exécute sa règle F après au plus $3n$ rondes. □

Le théorème suivant montre que tout parcours initié par *r* est un parcours en profondeur d'abord du réseau.

Théorème 5.4.4 *À partir de n'importe quelle configuration où* r *exécute la règle F, $\mathcal{DFS}1$ exécute un parcours en profondeur du réseau.*

Preuve. Supposons que le système démarre à partir d'une configuration γ_α où $\forall p \in V$, $S_p = C$. À partir de γ_α, *r* crée un jeton par sa règle F et ce jeton est unique dans réseau. Ensuite, comme expliqué dans la section 5.4.1, pendant tout le parcours, le système vérifie l'invariant suivant : l'ensemble $Visited$ du processeur qui détient le jeton contient les identités de tous les processeurs visités par le jeton. Ensuite, à chaque exécution des règles F et B, le processeur qui détient le jeton connait, grâce $Visited_p$, qui a été visité parmi ses voisins et, via la macro $Next_p$, soit p affecte D à S_p car tous ses voisins ont déjà été visité (dans ce cas, leurs identités sont dans $Visited_p$) soit p affecte q où q le voisin non-visité minimum dans l'ordre local $\prec_p$ (*i.e.*, le voisin minimum de p dans l'ordre local $\prec_p$, q, tel que $Id_q \notin Visited_p$) à S_p. D'où, à partir de γ_α, un parcours en profondeur du réseau est réalisé à partir de *r* en suivant la méthode présentée dans la remarque 5.3.1.

Supposons maintenant que le système démarre à partir d'une configuration quelconque (en particulier, une configuration différente de γ_α). Dans ce cas, le système peut contenir des processeurs pré-nettoyés et des PAPs anormaux. Cependant, nous allons maintenant montrer que ces processeurs pré-nettoyés et ces PAPs anormaux peuvent seulement ralentir la progression du PAP normal, $\mathcal{P}$, créé lors de l'exécution de la règle F de *r* : en fait, nous allons montrer qu'en dépit de ces objets (processeurs pré-nettoyés et PAPs anormaux), $\mathcal{P}$ évolue dans le réseau de la même manière que s'il avait été créé à partir de la configuration γ_α. Pour cela, supposons que *r* exécute sa règle F dans $\gamma \mapsto \gamma'$. Par la règle F, *r* désigne comme successeur son voisin minimal dans l'ordre local $\prec_r$ (en effet, grâce à $\neg LockedF(r)$, *r* exécute F seulement si $\forall p \in Neig_r$, $S_p = C$). Ensuite à chaque configuration atteinte à partir de γ' (γ' incluse), deux cas sont possibles en fonction de $S_{FE(P)}$:

- $S_{FE(\mathcal{P})} = p$, *i.e.*, p est le successeur de $FE(\mathcal{P})$. Dans ce cas, aucun processeur dans $\mathcal{P}$ est activable tant que p ne s'accroche pas à $\mathcal{P}$ (règle F). Ainsi, $\mathcal{P}$ est en "attente" jusqu'à ce que p s'accroche. Cependant, p ne peut s'accrocher à $\mathcal{P}$ tant que l'une de ces conditions est satisfaite :
  1. p a plusieurs prédécesseurs,
  2. $S_p \neq C$,
  3. p a un seul prédécesseur mais un de ses voisins non-visités q vérifie $S_q \neq C$. (n.b., par $PredVisited_p$, p connait l'ensemble de ses voisins visités par $\mathcal{P}$, lemme 5.4.4)

  Clairement, ces conditions sont satisfaites à cause de l'existence de processeurs pré-nettoyés et/ou de PAPs anormaux. Or, nous savons que, d'une part, les processeurs pré-nettoyés se nettoient en un temps fini (leur règle C est continument activable, cf. lemme 5.4.7) et, d'autre

part, on sait que le système ne contient plus de PAP anormal en un temps fini (lemme 5.4.1). Donc, p finit par s'accrocher à $\mathcal{P}$ et p choisit (en utilisant l'ordre local $\prec_p$) un successeur parmi ses voisins non-visités (*i.e.*, des voisins q tels que $Id_q \notin Visited_p$), si un tel processeur existe encore, sinon il affecte D à S_p (cf. macro $Next_p$).

– $S_{FE(\mathcal{P})} = D$. Quand $S_{FE(\mathcal{P})} = D$, cela signifie que le parcours à partir de $FE(\mathcal{P})$ est terminé et le père de $FE(\mathcal{P})$, q, doit continuer le parcours en désignant un nouveau successeur, si possible, en utilisant sa règle B. Or, dans ce cas, q est le seul processeur de $\mathcal{P}$ qui peut exécuter une règle et q attend jusqu'à ce que tous ses voisins non-visités satisfassent $S = C$ (grâce à $ChildVisited_q$, q connait l'ensemble de ses voisins visités par $\mathcal{P}$, lemme 5.4.4). D'où, comme pour le cas précédent, q exécute B en un temps fini et, par la règle B, q choisit, en fonction de $\prec_q$, un voisin non-visité comme successeur si de tels processeurs existent encore, sinon il affecte D à S_q (cf. la macro $Next_q$).

Ainsi, en démarrant à partir d'une configuration quelconque et en dépit des processeurs pré-nettoyés et des PAPs anormaux qui peuvent exister, le PAP normal $\mathcal{P}$ évolue dans le réseau de la même manière que s'il avait démarré à partir de la configuration γ_α et le théorème est vérifié. □

D'après la remarque 3.2.1 et les théorèmes 5.4.3 et 5.4.4, nous pouvons déduire le théorème suivant :

Théorème 5.4.5 *$\mathcal{DFS}1$ est un protocole instantanément stabilisant de parcours en profondeur sous un démon distribué faiblement équitable.*

Preuve de la stabilisation instantanée en supposant un démon distribué inéquitable. Nous avons montré que *$\mathcal{DFS}1$* est un protocole instantanément stabilisant de parcours en profondeur sous un démon faiblement équitable (théorème 5.4.5). Donc, pour prouver la stabilisation instantanée du protocole sous l'hypothèse d'un démon inéquitable, il reste à montrer que chaque parcours exécuté par notre protocole est réalisé en un nombre fini de mouvements (cf. premier paragraphe de la sous-section 5.4.2).

Pour cela, nous allons tout d'abord montrer que chaque PAP anormal disparaît après l'exécution d'un nombre fini d'accrochages et de décrochages.

Lemme 5.4.10 *Soit $\mathcal{P}$ un PAP anormal. $\mathcal{P}$ est* mort *après l'exécution d'au plus $2n-1$ actions d'accrochage et de décrochage.*

Preuve. Soit $e \in \mathcal{E}$ une exécution. Soit γ_i une configuration de e. Supposons qu'il existe un PAP anormal $\mathcal{P}$ dans γ_i. D'après l'algorithme, les trois types de règle peuvent être exécutés sur $\mathcal{P}$. La règle F permet à un processeur de s'accrocher à $\mathcal{P}$. L'exécution de la règle B provoque le décrochage de l'extrémité finale de $\mathcal{P}$ ($EF(\mathcal{P})$) : en effet, à tout moment, seul le prédécesseur p de l'extrémité finale d'un PAP peut exécuter sa règle B et, par cette règle, soit il exécute $S_p := D$ soit il désigne un nouveau successeur. Enfin, la règle C supprime l'extrémité initial du PAP.

Tout d'abord, nous avons montré dans la preuve du lemme 5.4.8 que le nombre de processeurs qui peuvent s'accrocher à $\mathcal{P}$ durant l'exécution est borné par $n - |\mathcal{P}|$ où $|\mathcal{P}|$ est la longueur de $\mathcal{P}$ dans γ_i.

Ensuite, dans le pire des cas, n processeurs doivent être enlevés de $\mathcal{P}$, *i.e.*, $|\mathcal{P}| + n - |\mathcal{P}|$ pour que $\mathcal{P}$ meure. Donc, au plus n règles B ou C sont exécutées sur $\mathcal{P}$ avant que $\mathcal{P}$ ne disparaisse.

D'où, dans le pire des cas, $\mathcal{P}$ est mort après $n - |\mathcal{P}|$ règles F (maximal pour $|\mathcal{P}| = 1$) et n règles B ou C, *i.e.*, $2n-1$ règles. □

Le théorème suivant montre qu'à partir de n'importe quelle configuration initiale la racine initie, via sa règle F, un parcours en un nombre fini de mouvements.

Théorème 5.4.6 *À partir de n'importe quelle configuration initiale,* r *exécute sa règle F en $O(n^2)$ mouvements.*

Preuve. Dans une configuration initiale, le système peut contenir :
- Un PAP normal non-initié par *r*.
- $O(n)$ PAPs anormaux.
- $O(n)$ processeurs pré-nettoyés.

Ensuite, chaque PAP (normal ou anormal) engendre $O(n)$ processeurs pré-nettoyés durant l'exécution. En effet, les processeurs pré-nettoyés engendrés par un PAP ont précédemment appartenu à ce PAP et nous savons, d'une part, qu'un PAP contient initialement $O(n)$ processeurs et que $O(n)$ processeurs s'accrochent à ce PAP durant l'exécution (lemme 5.4.6).

Donc, dans le pire des cas, à partir de n'importe quelle configuration initiale, avant que *r* exécute sa règle F :
- $O(n)$ règles sont exécutées sur le PAP normal non-initié par *r*, d'après le lemme 5.4.8.
- $O(n)$ règles sont exécutées sur chaque PAP anormal ($O(n)$ PAPs), d'après le lemme 5.4.10, *i.e.*, $O(n^2)$ règles.
- Une règle C pour chaque proceseurs pré-nettoyés, *i.e.*, $O(n)$ règles pour nettoyer les proceseurs pré-nettoyés présents dans la configuration initiale et $O(n^2)$ règles pour nettoyer les proceseurs pré-nettoyés engendrés par les PAPs ($O(n)$ processeurs pré-nettoyés engendrés par chacun des $O(n)$ PAP)

Ainsi, dans le pire des cas, à partir de n'importe quelle configuration initiale, $O(n^2)$ mouvements sont exécutés avant que *r* n'exécute sa règle F. □

Corollaire 5.4.1 *À partir de n'importe quelle configuration initiale, le protocole $\mathcal{DFS}1$ exécute un parcours en profondeur (complet) en $O(n^2)$ mouvements.*

D'après les explications fournies dans le premier paragraphe de la sous-section 5.4.2, le théorème 5.4.5 et le corollaire 5.4.1, nous pouvons déduire le théorème suivant :

Théorème 5.4.7 *$\mathcal{DFS}1$ est un protocole instantanément stabilisant de parcours en profondeur sous un démon distribué inéquitable.*

5.4.3 Complexité

Complexité en espace. D'après les variables déclarées dans les algorithmes 5.4.3 et 5.4.4, nous pouvons déduire le théorème suivant :

Théorème 5.4.8 *L'occupation mémoire du protocole $\mathcal{DFS}1$ est en $O(n \times \log n)$ bits par processeur.*

Il faut noter que le surcoût en mémoire de notre solution est important (*i.e.*, le rapport entre la complexité en espace de notre protocole et celle du protocole non tolérant aux fautes le plus efficace). En effet, un procotole non tolérant aux fautes de parcours en profondeur a besoin de $\Theta(\Delta)$ bits par processeur : chaque processeur doit pouvoir localement distinguer le successeur courant parmi ses voisins car le parcours est séquentiel (cf. sous-section 5.1.2, page 42). Le surcoût en mémoire du protocole *$\mathcal{DFS}1$* est donc en $O(n \times \log n)$.

Complexité en temps. Nous analysons maintenant la complexité en temps du protocole $\mathcal{DFS}1$ en fonction des critères présentés dans la sous-section 3.2.3.

Temps de stabilisation. D'après le théorème 5.4.7, $\mathcal{DFS}1$ est instantanément stabilisant. Donc, par la définition 3.2.1, le temps de stabilisation de $\mathcal{DFS}1$ est zéro à la fois en nombre de rondes et en nombre de mouvements.

Délai. D'après les théorèmes 5.4.3 et 5.4.6, nous savons que le délai du protocole $\mathcal{DFS}1$ est en $O(n)$ rondes et $O(n^2)$ mouvements, respectivement.

Le nombre de fois où le protocole doit être initié pour exécuter un parcours conforme aux spécifications. Par définition, pour n'importe quel protocole instantanément stabilisant, ce nombre est égal à un. Or, d'après le théorème 5.4.7, $\mathcal{DFS}1$ est instantanément stabilisant.

Le nombre de fois où un processeur peut participer à un parcours corrompu. D'après le lemme 5.4.6, chaque processeur peut s'accrocher au plus une fois à chaque PAP anormal. Or, initialement, le système contient $O(n)$ PAPs anormaux. Donc, durant l'exécution un processeur peut participer à $O(n)$ parcours corrompus.

Le nombre de fois où un processeur peut participer à un parcours corrompu sans le détecter comme tel. Dans le pire des cas, à chaque fois qu'un processeur participe à un parcours corrompu, il peut quitter ce parcours sans utiliser la règle de correction d'erreur (la règle C). Donc, en fonction du résultat précédent, chaque processeur peut participer à $O(n)$ parcours corrompus sans les détecter comme tels.

Le temps d'exécution nécessaire pour effectuer le premier parcours conforme aux spécifications. D'après le théorème 5.4.7, nous savons qu'à partir de n'importe quelle configuration initiale, le premier parcours initié par la racine sera conforme aux spécifications. Or, d'après le corollaire 5.4.1, nous savons qu'à partir de n'importe quelle configuration initiale, le protocole $\mathcal{DFS}1$ exécute un parcours en profondeur (complet) en $O(n^2)$ mouvements. Le théorème suivant nous montre que le temps d'exécution du premier parcours (conforme aux spécifications) est asymptotiquement optimal en nombre de rondes : $O(n)$ rondes. Il se déduit du lemme 5.4.9 et des théorèmes 5.4.1 et 5.4.3.

Théorème 5.4.9 *À partir de n'importe quelle configuration initiale, le protocole $\mathcal{DFS}1$ exécute un parcours en profondeur (complet) en au plus $5n - 1$ rondes.*

Le temps d'exécution nécessaire pour effectuer les parcours suivants. Comme nous l'avons affirmé dans la sous-section 3.2.3, de manière générale, pour un protocole stabilisant (auto-stabilisation ou stabilisation instantanée), le pire des cas du temps d'exécution d'un cycle (complet) de calcul conforme aux spécifications correspond au pire des cas de l'exécution du premier cycle conforme aux spécifications. C'est le cas aussi avec le protocole $\mathcal{DFS}1$. En effet, le temps d'exécution d'un parcours passe de $O(n^2)$ (corollaire 5.4.1) à $O(n)$ mouvements (théorème 5.4.10) (en revanche, la complexité en nombre de rondes reste identique). Cela est du au fait qu'après le premier parcours initié par la racine, le système ne contient plus de PAPs anormaux (lemme 5.4.11).

Lemme 5.4.11 *Après le premier parcours initié par la racine, le système ne contient plus de PAPs anormaux.*

Preuve. Supposons, par contradiction, que quelques PAPs anormaux persitent dans le système après le premier parcours initié par la racine. Soit $\mathcal{P}$ l'un de ces PAPs. Soit $\mathcal{P}'$ le premier PAP normal créé par *r*.

– Supposons que $\mathcal{P}$ est bloqué, *i.e.*, plus aucun processeur ne s'accroche à $\mathcal{P}$ durant l'évolution de $\mathcal{P}'$. Alors, comme $\mathcal{P}'$ visite tous les processeurs (par le théorème 5.4.7, $\mathcal{DFS}1$ vérifie sa spécification), $\mathcal{P}'$ fini par forcer les processeurs de $\mathcal{P}$ à se décrocher afin de les visiter. D'où, $\mathcal{P}$ finit par disparaître, contradiction.
– Supposons maintenant que quelques processeurs s'accrochent à $\mathcal{P}$ pendant le premier parcours initié par *r*.
Supposons ensuite que seuls des processeurs qui n'ont pas déjà été visités par $\mathcal{P}'$ s'accrochent à $\mathcal{P}$. Comme ce nombre de processeurs décroît (par le théorème 5.4.7, $\mathcal{DFS}1$ vérifie sa spécification), $\mathcal{P}$ finit par ne plus progresser dans le réseau. Donc, comme dans le cas précédent, $\mathcal{P}'$ finit par forcer les processeurs à se décrocher de $\mathcal{P}$ afin de les visiter et $\mathcal{P}$ finit par disparaître, contradiction.
Donc, quelques processeurs déjà visités par $\mathcal{P}'$ finissent par s'accrocher à $\mathcal{P}$. Soit q le premier processeur visité par $\mathcal{P}'$ qui s'accroche à $\mathcal{P}$. Supposons que q s'accroche à $\mathcal{P}$ dans le mouvement $\gamma \mapsto \gamma'$. Par la garde de la règle F, on sait que q satisfait $S_q = C$ dans γ. Donc, q s'est décroché de $\mathcal{P}'$ pour ensuite vérifier $S_q = C$. Or, q satisfait $S_p = D$ lorsqu'il se décroche de $\mathcal{P}'$ (comme $\mathcal{P}'$ est un PAP normal, q se décroche de $\mathcal{P}'$ lorsque son prédécesseur exécute la règle B). De plus, q exécute $S_q := D$ si et seulement si tous ses voisins ont été visités par $\mathcal{P}'$ (cf. les règles F et B et le lemme 5.4.4). D'où, dans γ, chaque voisin de q a déjà été visité par $\mathcal{P}'$. Or, dans γ, $\forall p \in \mathcal{P}$, p n'a pas été visité par $\mathcal{P}'$ (par hypothèse, q sera le premier visité dans $\gamma \mapsto \gamma'$). En particulier, $FE(\mathcal{P})$ n'a pas été visité par $\mathcal{P}'$ dans γ pourtant $FE(\mathcal{P}) \in Neig_q$, contradiction.

□

D'après les lemmes 5.4.8 et 5.4.11, on obtient :

Théorème 5.4.10 *Après le premier parcours initié par la racine, les autres parcours initiés sont exécutés en $O(n)$ mouvements.*

Le surcoût en temps. Pour évaluer le surcoût en temps de notre protocole, nous avons besoin de connaître :

(1) D'une part, le temps d'exécution d'un parcours à partir d'une configuration initiale normale. Cette complexité est en $O(n)$ rondes (lemme 5.4.9) et $O(n)$ mouvements (lemme 5.4.8).
(2) D'autre part, le temps d'exécution d'un parcours effectué dans un système sans faute par le protocole le plus efficace. Or, nous avons vu dans la section 5.3 que cette complexité est identique en terme de mouvements et de rondes : $2(n-1)$.

En tenant compte de ces deux complexités, nous pouvons déduire que le surcoût en temps de notre protocole (*i.e.* le rapport en (1) et (2)) est une constante en ce qui concerne les complexités à la fois en rondes et en mouvements.

5.4.4 Conclusion

Dans cette section, nous avons présenté le premier protocole de parcours en profondeur instantanément stabilisant pour un réseau enraciné quelconque et fonctionnant avec un démon distribué inéquitable. Ce protocole n'utilise pas de structures sous-jacentes telles que, par exemple, un arbre couvrant et n'a pas besoin de connaissances globales sur le réseau telles que sa taille, par exemple. Cependant, l'inconvénient majeur de ce protocole est qu'il utilise des listes d'identités. De ce fait, l'occupation mémoire de ce protocole est importante : $O(n \times \log n)$ bits par processeur. En revanche, notre protocole a de nombreux avantages. Par exemple, la propriété de stabilisation instantanée assure que dès que la racine initie un nouveau parcours, tous les processeurs sont visités dans un ordre

induit par la profondeur d'abord. Ensuite, l'analyse de complexité du protocole montre qu'il est très efficace en temps. Par exemple, il exécute un parcours complet en $O(n)$ rondes et $O(n^2)$ mouvements, respectivement. De plus, après le premier parcours initié, les autres parcours sont effectués en $O(n)$ mouvements. De plus, le surcoût en temps de notre protocole est négligeable : en effet, à partir d'une configuration initiale normale, son temps d'exécution est quasiment identique à celui du protocole non tolérant aux fautes le plus efficace. Enfin, nous pouvons remarquer que tout parcours initié par la racine visite toujours les noeuds dans le même ordre : en fait, le parcours en profondeur réalisé par notre protocole est dit *premier* comme défini par Collin et Dolev dans [CD94]. En effet, grâce aux liste d'identités, un processeur p, lorsqu'il détient le jeton initié par la racine, désigne toujours comme successeur son voisin non-visité minimal dans l'ordre local $\prec_p$ (si un tel processeur existe). L'arbre couvrant construit lors de la première exécution de $\mathcal{DFS}1$ est donc un point fixe.

5.5 Deuxième solution

L'occupation mémoire est l'inconvénient majeur de la solution précédente. La complexité en espace du protocole $\mathcal{DFS}1$ est due à l'utilisation de listes d'identités. Dans cette section, nous présentons une autre solution au problème du parcours en profondeur instantanément stabilisant dans un réseau enraciné quelconque. Le protocole proposé est significativement meilleur en complexité en espace que le protocole $\mathcal{DFS}1$ et n'utilise plus les identités des processeurs. Dans le protocole $\mathcal{DFS}1$, les listes d'identités sont principalement utilisées pour permettre à chaque processeur du parcours initié par la racine de détecter quand tous ses voisins ont été visités par le parcours. Cette propriété de détection est indispensable pour obtenir un protocole instantanément stabilisant. Dans cette deuxième solution, nous avons remplacé les listes d'identités par un mécanisme de question : lorsqu'un processeur reçoit un jeton, il questionne dynamiquement la racine (normale ou anormale) de son parcours afin de savoir si lui et ses voisins sont dans un parcours issu de la racine du réseau (*r*). Ce mécanisme a été précédemment introduit par Blin *et al* dans [BCV03]. Cette deuxième solution a été présentée lors de la conférence internationale *SSS'05* ([CDV05b]).

Nous allons maintenant présenter informellement notre deuxième solution, appelée protocole $\mathcal{DFS}2$ (sous-section 5.5.1). Nous proposerons ensuite la preuve de stabilisation instantanée du protocole (sous-section 5.5.2). Puis, nous discuterons de sa complexité (sous-section 5.5.3) avant de conclure (sous-section 5.5.4).

5.5.1 Le protocole

Le protocole $\mathcal{DFS}2$ est présenté dans les algorithmes 5.5.5 et 5.5.6. Dans cette deuxième solution, la *phase de visite* diffère en deux points du protocole de base ($\mathcal{DFS}0$, page 47) : nous utilisons une variable de niveau pour détecter les cycles possibles sur les pointeurs père et fils et une phase dite de *question*. Les questions sont émises en direction de la racine par un processeur recevant le jeton afin de confirmer son appartenance au parcours normal ainsi que celle de ses voisins *a priori* visités. La réponse attendue ne peut être dynamiquement engendrée que par *r*, assurant ainsi en cas de réponse positive que les processeurs concernés sont bien dans le parcours normal. Dès lors, le processeur à l'origine de la question est assuré qu'il peut renvoyer le jeton à son père sans avoir tronqué le parcours.

La *phase de nettoyage* est plus complexe que dans le protocole de base ($\mathcal{DFS}0$, page 47) afin de ne pas engendrer une complexité trop importante dûe à la possibilité du parcours enraciné en *r* de visiter plusieurs fois les mêmes processeurs si ce parcours était déjà présent dans la configuration initiale.

La *phase de correction* quant à elle est exécutée en deux temps : dans un premier temps un mécanisme gèle la progression du jeton d'un parcours anormal à partir de sa racine anormale. Une fois cette progression gelée, les processeurs de ce parcours peuvent alors se corriger à partir de la racine. Le gel du parcours se fait par l'utilisation d'un PIR sur l'arbre correspondant à ce parcours.

Nous décrivons maintenant ces quatre phases plus en détails.

Phase de visite. À partir d'une configuration initiale quelconque, les variables de la phase de visite de $\mathcal{DFS}0$ peuvent former des cycles de successeurs (un exemple de cycle de successeurs est montré dans la figure 5.5.8). Dans $\mathcal{DFS}2$, nous détectons cette erreur en utilisant une variable supplémentaire L. Pour la racine, L_r est une constante égale à zéro. Les autres processeurs p calculent leur variable L_p à chaque fois qu'il recoivent un nouveau jeton (règle F) comme suit : $L_p := L_q + 1$ où q est l'unique prédécesseur de p (n.b., la règle F de p est activable seulement si p n'a qu'un prédécesseur). En fait, L_p doit être égale à la longueur du chemin élémentaire de *r* à p dans l'arbre

Algorithm 5.5.5 Algorithme $\mathcal{DFS}2$ pour p = *r*

Entrée-sortie : $Request_p \in \{Wait,In,Out\}$;

Entrées : $Neig_p$: ensemble des voisins (localement ordonnés) ;

Constantes : $L_p = 0$; $Par_p = \perp$;

Variables : $S_p \in Neig_p \cup \{C,D,PC\}$; $Que_p \in \{Q,R,A\}$;

Macros :

$Next_p = (q = \min_{\prec_p}\{q' \in Neig_p :: S_{q'} = C\})$ **si** q **existe**, D **sinon** ;

$Child_p = \{q \in Neig_p :: Par_q = p \wedge S_q \neq C \wedge L_q = L_p + 1 \wedge [(S_p = q) \Rightarrow (S_q \notin \{PC,EB,EF\})] \wedge [[S_p \neq q] \Rightarrow [(S_p = PC \wedge S_q = PC) \vee (S_q = D)]]\}$;

Prédicats généraux :

$End(p) \equiv [\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Par_q \neq p)]$

$AnswerOK(p) \equiv (Que_p = A) \wedge [\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q = A)]$

Gardes :

$Forward(p) \equiv (S_p = C) \wedge (\forall q \in Neig_p :: S_p = C)$

$Backward(p) \equiv (\exists q \in Neig_p :: S_p = q \wedge Par_q = p \wedge S_q = D) \wedge [AnswerOK(p) \vee (\exists q \in Neig_p :: S_q = C)]$

$PreClean(p) \equiv (S_p = D) \wedge (S_{Par_p} = PC) \wedge [\forall q \in Neig_p :: (Par_q = p) \Rightarrow (S_q \in \{C,D\})]$

$Clean(p) \equiv (S_p = PC) \wedge End(p)$

$Require(p) \equiv (S_p \neq C) \wedge [[Que_p = Q \wedge (\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q \in \{Q, R\}))] \vee [Que_p = A \wedge (\exists q \in Neig_p :: (S_q \neq C \wedge Que_q = Q) \vee (q \in Child_p \wedge Que_q = R))]]$

$Answer(p) \equiv (S_p \neq C) \wedge (Que_p = R) \wedge (\forall q \in Child_p :: Que_q \in \{W,A\}) \wedge [\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q \neq Q)]$

Règles :

Phase de Visite :

F	::	$Forward(p) \wedge (Request_p = Wait)$	$\rightarrow$	$S_p := Next_p$; $Que_p := Q$; $Request_p := In$;	*/* Démarrage */*
B	::	$Backward(p)$	$\rightarrow$	$S_p := Next_p$;	

Phase de Nettoyage :

PC	::	$PreClean(p)$	$\rightarrow$	$S_p := PC$;	
C	::	$Clean(p)$	$\rightarrow$	$S_p := C$;	
T	::	$Forward(p) \wedge (Request_p = In)$	$\rightarrow$	$Request_p := Out$;	*/* Terminaison */*

Phase de Question :

R	::	$Require(p)$	$\rightarrow$	$Que_p := R$;	
A	::	$Answer(p)$	$\rightarrow$	$Que_p := A$;	

couvrant calculé lors du parcours. Évidemment, dans un cycle de successeurs, au moins un processeur p vérifie $L_p \neq L_{Par_p} + 1$ (cf. figure 5.5.8).

Ensuite, nous avons vu que dans $\mathcal{DFS}0$, un processeur p détenant le jeton et constatant que $\forall p' \in Neig_p$, $S_{p'} \neq C$ renvoie le jeton à son père : en effet, dans ce cas, tous ses voisins ont été visités par ce jeton. Ici, cette condition n'est pas suffisante pour affirmer que tous les voisins de p ont été visités car le système peut contenir plusieurs jetons dûs à la configuration initiale quelconque. Pour garantir cette propriété lorsque p renvoie le jeton à son père, nous introduisons une nouvelle phase : la *phase de question*. Le but de cette phase est de garantir qu'un processeur p participant au parcours initié par *r* exécute $S_p := D$ seulement quand tous ses voisins ont été visités par le parcours. Pour appliquer cette méthode, nous avons tout d'abord ajouté l'état P (*Pause*) dans la définition de S_p, $\forall p \in V \setminus \{r\}$. Nous modifions alors la règle F comme suit : quand un processeur $p \neq r$ reçoit un nouveau jeton et qu'il vérifie $\forall p' \in Neig_p$, $S_{p'} \neq C$, il affecte maintenant P à S_p au lieu de D (macros $Next_p$ et $PorD_p$). Cette dernière affectation initie une phase de question que nous décrivons ci-dessous.

Phase de question. Pour réaliser cette phase, nous avons besoin d'une variable supplémentaire : Que_p, $Que_p \in \{Q,R,W,A\}$ pour $p \neq r$ et $Que_p \in \{Q,R,A\}$ pour $p = r$. Les valeurs Q et R sont utilisées pour réinitialiser les variables Que dans la partie du réseau qui est concernée par la question.

Algorithm 5.5.6 Algorithme $\mathcal{DFS}2$ pour $p \neq r$

Entrées : $Neig_p$: ensemble des voisins (localement ordonnés) ;

Variables : $S_p \in Neig_p \cup \{C,P,D,PC,EB,EF\}$; $Par_p \in Neig_p$; $L_p \in \mathbb{N}$; $Que_p \in \{Q, R, W, A\}$;

Macros :

$PorD_p$ = P **si** $(S_p = C)$, D **sinon** ;
$Next_p$ = $(q = \min_{\prec_p}\{q' \in Neig_p :: S_{q'} = C\})$ **si** q **existe**, $PorD_p$ **sinon** ;
$Pred_p$ = $\{q \in Neig_p :: S_q = p\}$;
$Child_p$ = $\{q \in Neig_p :: Par_q = p \wedge S_q \neq C \wedge L_q = L_p + 1 \wedge (S_p = q \Rightarrow S_q \notin \{PC,EB,EF\})$
$\wedge [[S_p \neq q] \Rightarrow [(S_p = EB) \vee (S_p = EF \wedge S_q = EF) \vee (S_p = PC \wedge S_q = PC)$
$\vee (S_p \in Neig_p \cup \{D,PC\} \wedge S_q = D)]]\}$;

Prédicats généraux :

$GoodLevel(p)$ $\equiv$ $(S_p \neq C) \Rightarrow (L_p = L_{Par_p} + 1)$
$GoodS(p)$ $\equiv$ $(S_p \neq C) \Rightarrow [[(S_{Par_p} = p) \Rightarrow (S_p \notin \{PC,EB,EF\})]$
$\wedge [(S_{Par_p} \neq p) \Rightarrow ((S_{Par_p} = EB) \vee (S_{Par_p} = EF \wedge S_p = EF)$
$\vee (S_{Par_p} = PC \wedge S_p = PC) \vee (S_{Par_p} \in Neig_p \cup \{D,PC\} \wedge S_p = D))]]$
$AbRoot(p)$ $\equiv$ $\neg GoodS(p) \vee \neg GoodLevel(p)$
$End(p)$ $\equiv$ $[\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Par_q \neq p)]$
$AnswerOK(p)$ $\equiv$ $(Que_p = A) \wedge (\forall q \in Neig_p :: S_q \neq C \Rightarrow Que_q = A)$

Gardes :

$EFAbRoot(p)$ $\equiv$ $(S_p = EF) \wedge AbRoot(p) \wedge [\forall q \in Neig_p :: (Par_q = p \wedge L_q > L_p) \Rightarrow (S_q \in \{EF,C\})]$
$EBroadcast(p)$ $\equiv$ $(S_p \notin \{C,EB,EF\}) \wedge [\neg AbRoot(p) \Rightarrow (S_{Par_p} = EB)]$
$EFeedback(p)$ $\equiv$ $(S_p = EB) \wedge [\forall q \in Neig_p :: (Par_q = p \wedge L_q > L_p) \Rightarrow (S_q \in \{EF,C\})]$
$Forward(p)$ $\equiv$ $(S_p = C) \wedge (|Pred_p| = 1) \wedge End(p)$
$POk(p)$ $\equiv$ $(S_p = P) \wedge [AnswerOK(p) \vee (\exists q \in Neig_p :: S_q = C)] \wedge End(p)$
$Backward(p)$ $\equiv$ $(\exists q \in Neig_p :: S_p{=}q \wedge Par_q{=}p \wedge S_q{=}D) \wedge [AnswerOK(p) \vee (\exists q \in Neig_p :: S_q{=}C)]$
$BadSucc(p)$ $\equiv$ $AnswerOk(p) \wedge (\exists q \in Neig_p :: S_p = q \wedge q \notin Child_p \wedge S_q \neq C)$
$PreClean(p)$ $\equiv$ $(S_p = D) \wedge (S_{Par_p} = PC) \wedge [\forall q \in Neig_p :: (Par_q = p) \Rightarrow (S_q \in \{C,D\})]$
$Clean(p)$ $\equiv$ $(S_p = PC) \wedge End(p)$
$Require(p)$ $\equiv$ $(S_p \notin \{C,EB,EF\}) \wedge [[Que_p = Q \wedge (\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q \in \{Q,R\}))]$
$\vee [Que_p \in \{W,A\} \wedge (\exists q \in Neig_p :: (S_q \neq C \wedge Que_q = Q)$
$\vee (q \in Child_p \wedge Que_q = R))]]$
$WaitAnswer(p)$ $\equiv$ $(S_p \notin \{C,EB,EF\}) \wedge (Que_p = R)$
$\wedge (Que_{Par_p} = R) \wedge (\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q \neq Q))$
$\wedge [End(p) \Rightarrow (S_p \neq PC \wedge ((S_p \in Neig_p) \Rightarrow (S_{S_p} \neq C)))]$
$\wedge [\neg End(p) \Rightarrow (\forall q \in Child_p :: Que_q \in \{W,A\})]$
$Answer(p)$ $\equiv$ $(S_p \notin \{C,EB,EF\}) \wedge (Que_p = W) \wedge (Que_{Par_p} = A)$
$\wedge (\forall q \in Child_p :: Que_q \in \{W,A\}) \wedge (\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q \neq Q))$

Règles :

Phase de Correction :

EC :: $EFAbRoot(p)$ $\rightarrow$ $S_p := C$;
EB :: $EBroadcast(p)$ $\rightarrow$ $S_p := EB$;
EF :: $EFeedback(p)$ $\rightarrow$ $S_p := EF$;

Phase de Visite :

F :: $Forward(p)$ $\rightarrow$ $Par_p := (q \in Pred_p)$; $S_p := Next_p$; $L_p := L_{Par_p} + 1$; $Que_p := Q$;
$Fbis$ :: $POk(p)$ $\rightarrow$ $S_p := Next_p$;
B :: $Backward(p)$ $\rightarrow$ $S_p := Next_p$;
IE :: $BadSucc(p)$ $\rightarrow$ $S_p := Next_p$;

Phase de Nettoyage :

PC :: $PreClean(p)$ $\rightarrow$ $S_p := PC$;
C :: $Clean(p)$ $\rightarrow$ $S_p := C$;

Phase de Question :

R :: $Require(p)$ $\rightarrow$ $Que_p := R$;
W :: $WaitAnswer(p)$ $\rightarrow$ $Que_p := W$;
A :: $Answer(p)$ $\rightarrow$ $Que_p := A$;

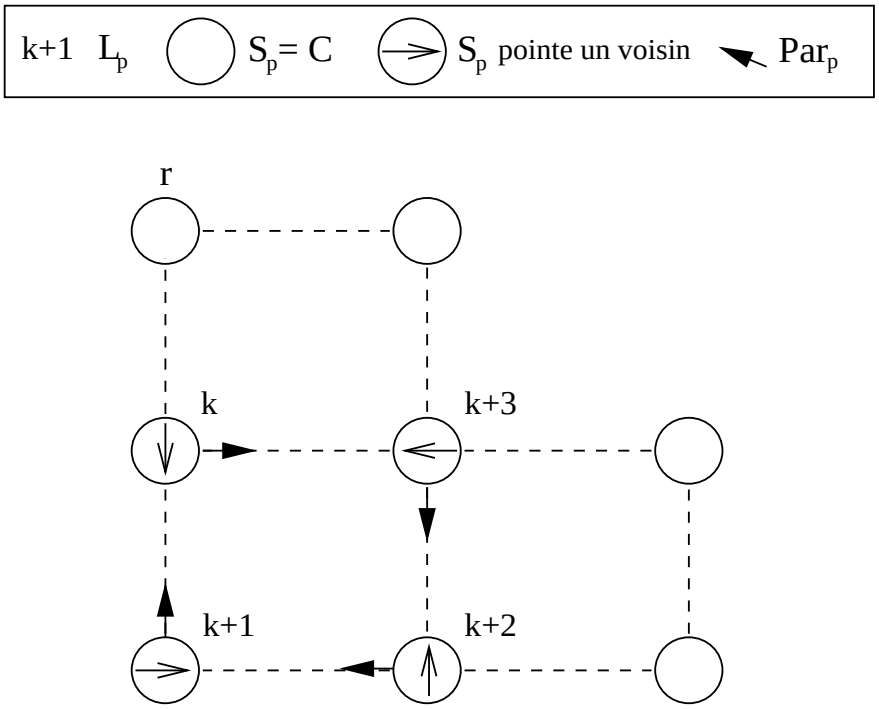


FIG. 5.5.8 – Exemple d'un cycle de successeurs.

La valeur W est générée à chaque fois qu'un processeur a besoin de savoir si ses voisins *a priori* visités appartiennent au parcours de la racine. La valeur A correspond à la réponse (positive) envoyée par la racine (n.b., la racine est l'unique processeur qui peut générer une réponse A). Expliquons maintenant comment fonctionne cette phase (n.b. la figure 5.5.9 illustre la phase de question par un exemple simple).

Tout d'abord, la phase de question ne concerne que les processeurs appartenant à un parcours, *i.e.*, les processeurs p tels que $S_p \neq C$. Ensuite, la variable Que_p est initialisé avec Q lorsque le processeur p exécute sa règle F. Cette valeur force tous les voisins de p tels que $S \neq C$ à exécuter $Que := R$ (règle R). Quand tous les voisins de p tels que $S_p \neq C$ ont réinitialisé leur variable Que, p exécute $Que_p := R$. Les valeurs R sont ensuite propagées dans le parcours de chacun de ces processeurs en suivant les pointeurs Par (règle R). Par ce mécanisme, les valeurs A sont effacées dans les chemins "pères" de p et de ses voisins tels que $S \neq C$ (en particulier, les valeurs A présentes dans le système depuis la configuration initiale). Ainsi, à partir de maintenant, quand une valeur A atteindra p ou l'un de ses voisins tels que $S \neq C$, cette valeur ne pourra venir que de *r* et nous serons sûre que le processeur appartient au parcours issu de *r*.

Comme nous l'avons vu précédemment, un processeur q ($q \neq r$) finit par attendre une réponse de *r* pour savoir si lui et ses voisins appartiennent à $Arbre(r)$, *i.e.*, $S_q = P$. Dans ce cas, q et ses voisins tels que $S = D$ exécutent leur règle W[2], *i.e.*, $Que := W$ signifiant qu'ils sont en attente d'une réponse de la part de *r*. Les valeurs W sont ensuite propagées jusqu'à *r* si possible (*i.e.*, dans le cas où elles sont dans $Arbre(r)$) comme suit : un processeur p propage une valeur W quand tous ses fils vérifient $Que \in \{W, A\}$ et que tous ses voisins vérifient $S \neq C \Rightarrow Que \neq Q$). Quand les valeurs W atteignent les fils de *r*, *r* exécute sa règle A : *r* diffuse une réponse (A) à partir de ses fils. Donc, si q et ses voisins recoivent cette réponse (A), q est sûr que lui et tous ses voisins appartiennent à $Arbre(r)$. Dans ce cas, q satisfait $AnswerOk(q)$ et exécute sa règle $Fbis$ pour affecter D à S_q et le parcours

[2]En démarrant d'une configuration quelconque, des voisins q' de q tels que $S_{q'} = D$ peuvent appartenir à $Arbre(r)$, donc, ils doivent aussi recevoir une réponse positive de *r* pour que q sache qu'ils ne sont pas dans un parcours anormal.

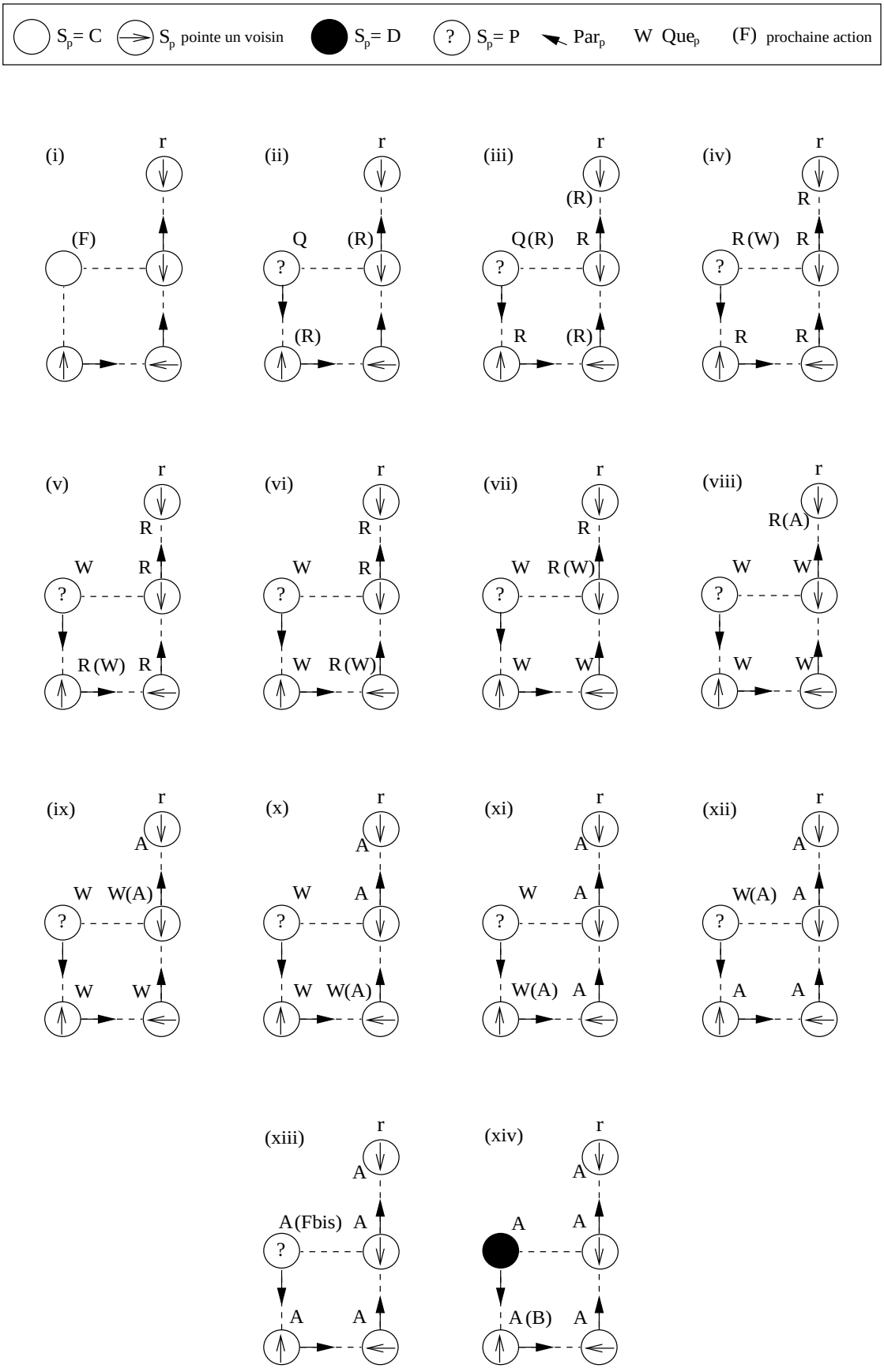


FIG. 5.5.9 – Exemple d'une phase de question.

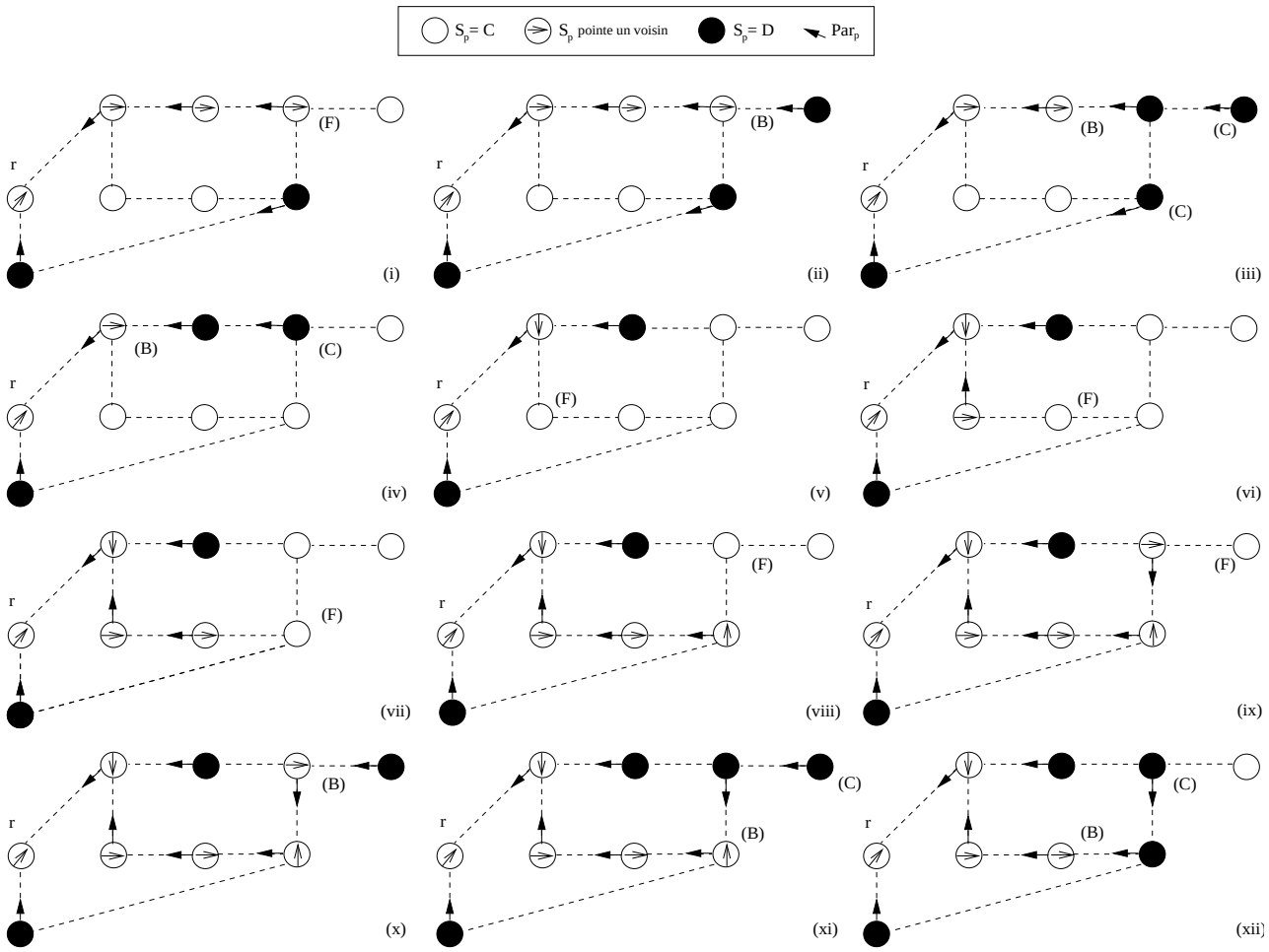


FIG. 5.5.10 – Problème du nettoyage direct à partir des feuilles.

peut reprendre à partir de son père, etc. Dans le cas contraire, le parcours reste bloqué jusqu'à ce que la phase de correction efface les parcours anormaux passant par q ou ses voisins. Supposons maintenant qu'un voisin de q, q', s'accroche à un parcours anormal après que q ait initié sa question. Alors, par sa règle F, q' exécute, en particulier, $Que_{q'} := Q$ et comme expliqué précédemment, tant que q' est dans un parcours anormal, $Que_{q'} \neq A$. D'où, lorsqu'un processeur q initie une phase de question (par $Que_q := Q$), il ne pourra par la suite exécuter $S_q := D$ que lorsque lui et ses voisins appartiendront à $Arbre(r)$.

Phase de nettoyage. Le but de la phase de nettoyage est d'effacer les traces du parcours précédent afin que le système puisse à nouveau démarrer (n.b. cette phase ne réinitialise pas les pointeurs Par). Utilisée à partir d'une configuration quelconque, la phase de nettoyage du protocole $\mathcal{DFS}0$ peut engendrer un surcoût important en temps d'exécution. En effet, à cause de ce nettoyage direct à partir des feuilles de l'arbre de parcours, le parcours de la racine peut visiter plusieurs fois les mêmes processeurs. La figure 5.5.10 nous montre un exemple d'exécution dans lequel le processeur le plus à droite est visité deux fois. En généralisant ce petit exemple, une partie importante du réseau peut être visitée plusieurs fois et nous obtenons un surcoût de $O(n^2)$ mouvements (resp. rondes) : par exemple, dans la figure 5.5.11, les processeurs entourés peuvent être visités plusieurs fois.

Pour résoudre ce problème, nous avons choisi de réaliser le nettoyage en deux étapes. Tout d'abord, nous attendons que *r* détecte la fin du parcours (n.b. *r* est le seul processeur qui peut détecter

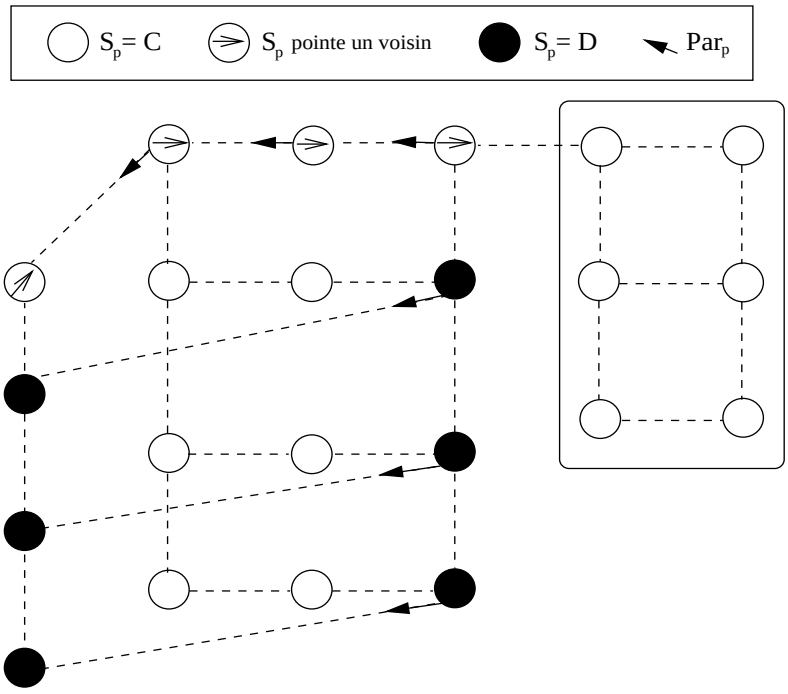


FIG. 5.5.11 – Généralisation de la figure 5.5.10.

lorsque le parcours normal est entièrement terminé : lorsque $S_r = D$) pour diffuser une valeur, PC (*PreClean*), dans $Arbre(r)$ à partir de *r* (cf. règle PC). Cette valeur a pour but d'informer tous les processeurs du parcours de la terminaison (à cet effet, nous ajoutons l'état PC dans la définition de S_p, $\forall p \in V$). Ensuite, l'arbre est nettoyé à partir des feuilles (règle C) en affectant C aux variables S. Ainsi, comme le parcours termine avant de se nettoyer, le nettoyage ne peut plus provoquer la re-visite de processeurs. Un exemple de phase de nettoyage en deux étapes est présenté dans la figure 5.5.12.

Nous allons maintenant traiter les configurations anormales. Tout d'abord, nous allons introduire une nouvelle règle pour la phase de visite : la règle IE. Cette règle traite un interblocage dû à la valeur initiale des variables de la phase de visite.

Règle spéciale : la règle IE. Le cas suivant ne peut apparaître que dans la configuration initiale : le processeur qui détient le jeton du parcours de *r* peut désigner comme successeur un voisin q (*i.e.*, $S_p = q$) tel que q appartient déjà à $Arbre(r)$ (une telle configuration est montrée figure 5.5.13). Cependant, grâce à la phase de question, p finit par détecter qu'il désigne un "mauvais" successeur car p et q reçoivent tous les deux une réponse (A) de la part de *r*. Afin de ne pas surcharger le protocole, nous choisissons de ne pas interrompre le parcours de *r* dans ce cas bien qu'ayant détecté une erreur évidente signifiant que ce parcours n'a pas été initié dynamiquement après la configuration initiale et n'est donc d'aucune utilité. Dans ce cas, p change simplement de successeur en exécutant sa règle IE. Dans la suite, nous considèrerons que la règle IE est une règle de la phase de visite.

Nous présentons maintenant la phase de correction. Cette phase permet d'effacer les parcours anormaux, c'est à dire, les parcours issus d'un processeur autre que *r*.

Phase de correction. Cette phase concerne uniquement les processeurs p tels que $S_p \neq C$ et $p \notin Arbre(r)$, c'est à dire, des processeurs appartenant à des parcours *anormaux*. Les parcours anormaux sont organisés en arbres dit *anormaux*. Ces arbres sont enracinés en un processeur dit *racine*

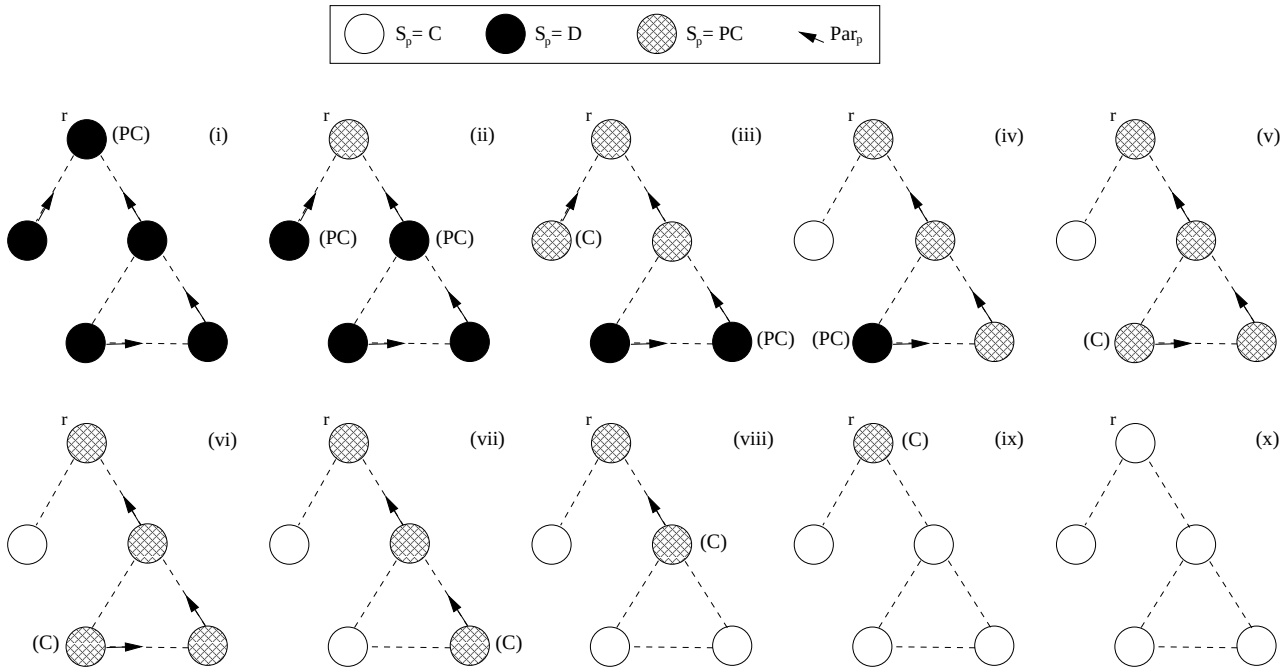


FIG. 5.5.12 – Exemple de nettoyage en deux temps.

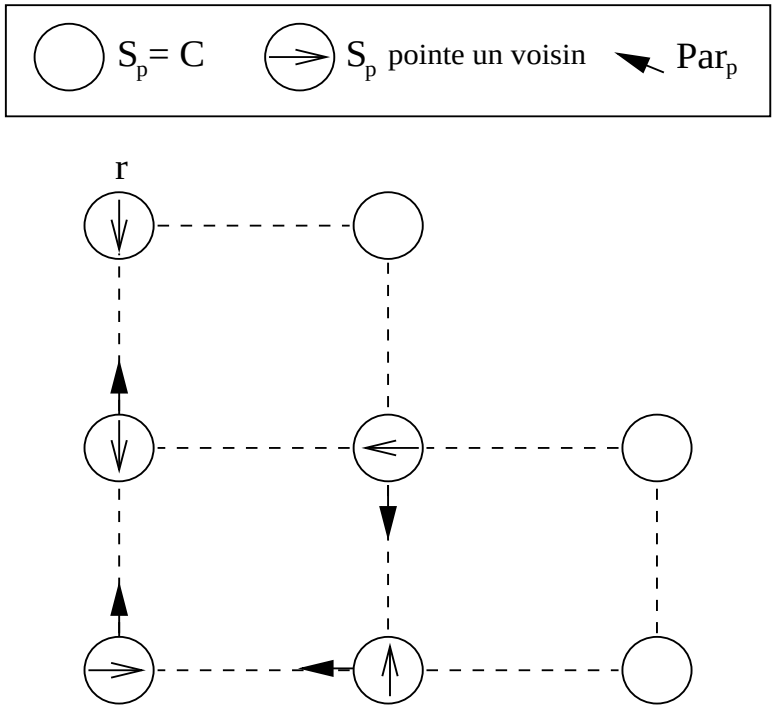


FIG. 5.5.13 – Exemple d'interblocage.

anormale. Une racine anormale est un processeur qui vérifie le prédicat $AbRoot$, *i.e.*, un processeur $p \neq r$ et vérifiant S_p dont l'état est incohérent vis à vis de celui de son père (Par_p) par rapport à la trace du parcours du jeton ou la trace du mécanisme de gel que nous allons détailler ensuite (en fait, p vérifie $AbRoot(p)$ lorsque l'état du couple (p,Par_p) ne peut pas être obtenu dynamiquement lors d'une exécution). La trace des parcours enracinés en ces racines anormales forme des arbres : un processeur q appartient à l'arbre anormal enraciné en p, $Arbre(p)$, (p vérifie $AbRoot(p)$) si et seulement si il existe une suite de processeurs ($p_0 = p$), ..., p_i, ..., ($p_k = q$) telle que, $\forall i \in [1...k]$, $(Par_{p_i} = p_{i-1}) \wedge (\neg AbRoot(p_i))$ (*i.e.*, il existe un chemin père de q à p où p est l'unique processeur vérifiant $AbRoot$).

La phase de correction consiste ainsi en la suppression de ces arbres anormaux. Pour supprimer de tels arbre anormaux, nous ne pouvons pas simplement les supprimer à partir de leur racine (l'unique processeur de l'arbre à savoir qu'il est dans un arbre anormal) jusqu'à leurs feuilles. En effet, pour supprimer l'arbre anormal $Arbre(p)$, si p a un successeur courant dans son parcours (*i.e.*, $S_p \notin \{P,D,PC\}$) et que nous affectons directement C à S_p, alors p peut à nouveau être visité par son ancien parcours qui se poursuit via son successeur. Or, comme nous ne supposons pas de borne sur les variables L (nous aurions pu supposer que la valeur maximale de L est une borne supérieure de n), ce cas peut se reproduire infiniment souvent et le système peut contenir des arbres anormaux qui peuvent bloquer la progression du parcours normal durant toute l'exécution. Nous résolvons ce problème en exécutant un PIR paralysant sur chaque arbre anormal avant de les effacer (cette méthode est illustrée par la figure 5.5.14). Pour appliquer cette méthode, les règles de la phase de correction utilisent deux états supplémentaires dans les variables S : EB (*ErrorBroadcast*) et EF (*ErrorFeedback*) pour tout processeur p tel que $p \neq r$. Ensuite, si p est une racine anormale, p affecte EB à sa variable S_p, cette valeur est alors diffusée dans son arbre et uniquement son arbre (règle EB). Cette valeur rend inactivable les règles des autres phases. Puis, la règle EF assure que p finit par recevoir (via ses fils) un accusé de la réception par tous processeurs de son arbre de la valeur EB (la valeur EF) ; p est alors sûr que tous les processeurs q de son arbre vérifient $S_q = EF$ et qu'aucun autre processeur ne peut plus être visité par son parcours. L'arbre $Arbre(p)$ peut alors être effacé à partir de p (règle EC). Ainsi, par ce mécanisme, tous les arbres anormaux finissent par disparaître du système.

5.5.2 Preuve de la stabilisation instantanée

Nous allons maintenant prouver que le protocole $\mathcal{DFS}2$ est un protocole instantanément stabilisant de parcours en profondeur sous l'hypothèse d'un démon distribué inéquitable. Tout d'abord, comme annoncé au début de ce chapitre, nous rappelons que dans les preuves la valeur de la variable $Request_r$ sera considérée comme transparente pour le protocole (la gestion explicite des requêtes sera traitée dans la section 5.6) : nous ne tiendrons pas compte de cette variable lors de l'évalution et de l'exécution des règles. Par exemple, nous supposerons que la garde de la règle F de r est simplement le prédicat $Forward(r)$. Ensuite, comme pour le protocole précédent, la preuve de la stabilisation instantanée du protocole $\mathcal{DFS}2$ sous un démon distribué inéquitable sera faite en deux étapes (conformément au théorème 4.3.1, page 35) :

(i) Tout d'abord, nous allons prouver que $\mathcal{DFS}2$ est un protocole instantanément stabilisant de parcours en profondeur en supposant un démon distribué faiblement équitable.

(ii) Puis, nous prouverons que chaque parcours effectué par $\mathcal{DFS}2$ ne peut s'exécuter qu'en un nombre fini de mouvements.

Avant de démontrer la stabilisation instantanée de notre protocole, nous allons définir des objets que nous utiliserons dans les preuves et prouver certaines de leurs caractéristiques.

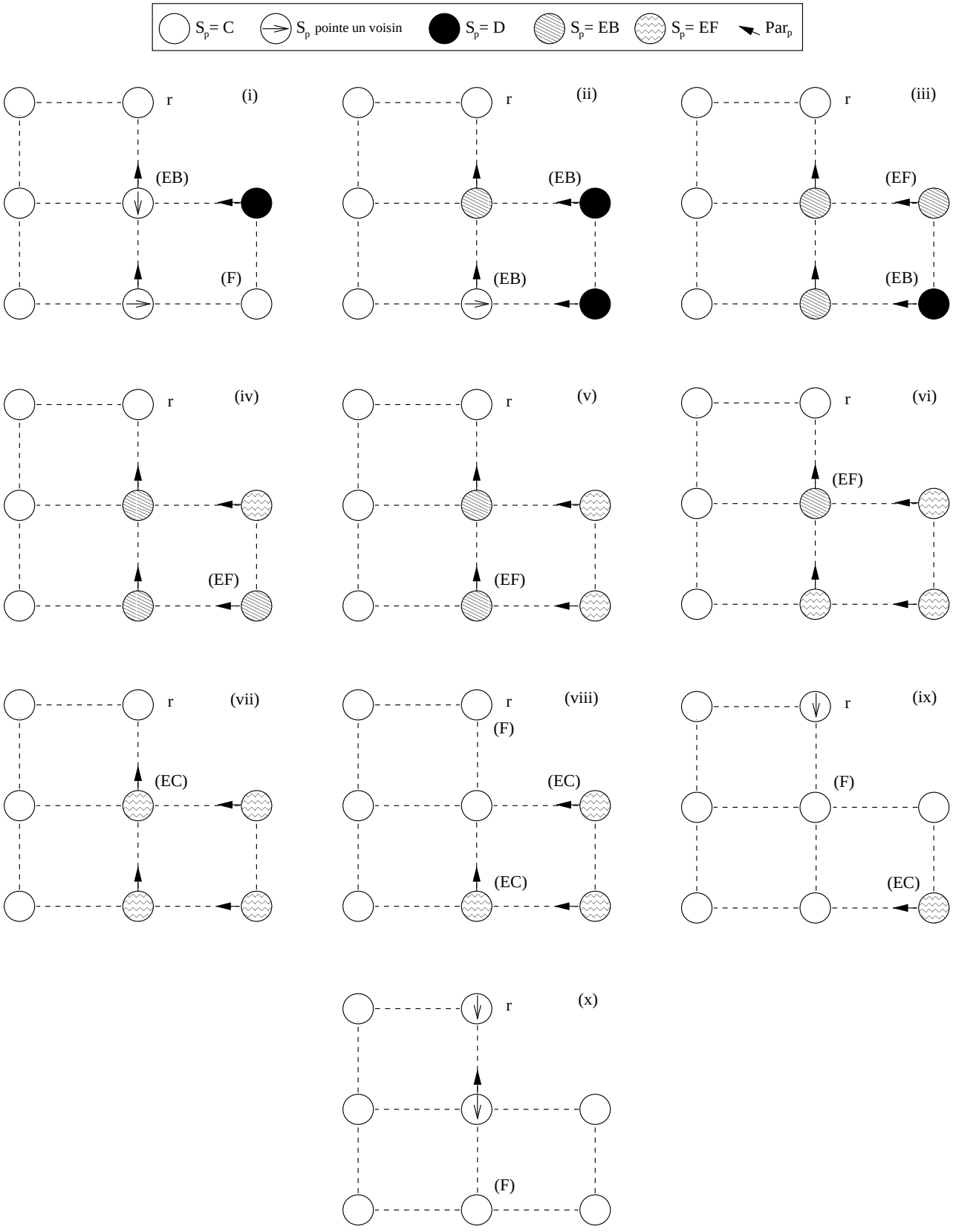


FIG. 5.5.14 – Exemple de phase de correction.

Définitions et propriétés. Dans la première définition, nous introduisons la notion de *chemin père*. Pour tout processeur p participant à un parcours, le chemin "père" de p est le chemin suivant les pointeurs Par qui relie p à la racine (normale ou anormale) de son parcours.

Définition 5.5.1 (Chemin Père) *$\forall p \in V$ tel que $S_p \neq C$, on appelle* chemin père *de p, noté dans la suite $CheminPere(p)$, l'unique chemin p_0,p_1,p_2,... ($p_k = p$) vérifiant les conditions suivantes :*

1. *$\forall i$, $1 \leq i \leq k$, ($S_{p_i} \neq C$) $\wedge$ ($Par_{p_i} = p_{i-1}$) $\wedge \neg AbRoot(p_i)$.*
2. *($p_0 =$ r) $\vee$ $AbRoot(p_0)$.*

Le concept d'arbre de parcours défini ci-dessous permet de représenter la trace laissée par un parcours dans le système.

Définition 5.5.2 (Arbre de parcours) *$\forall p \in V$ tel que ($p =$ r) $\vee$ $AbRoot(p)$, nous appelons* arbre de parcours *de p (dans la suite, on dira plus simplement arbre de p et p sera appelé* racine *de l'arbre) l'ensemble de processeurs $Arbre(p)$ défini comme suit : $\forall q \in V$, $q \in Arbre(p)$ si et seulement si $S_q \neq C$ et p est l'extrémité initiale de $CheminPere(q)$.*

Dans la remarque suivante, nous rappelons quelques notions de la théorie des graphes que nous avons adaptées à notre structure d'arbre de parcours.

Remarque 5.5.1 *Soit T un arbre. Soit $p \in T$ et $p_0, p_1, p_2, \ldots$ ($p_k = p$) son* chemin père. *$\forall i$, $1 \leq i \leq k$, p_{i-1} est dit* père *de p_i dans T. Inversement, p_i est considéré comme un* fils *de p_{i-1} dans T. Nous appelons* hauteur *de p_i dans T la longueur du chemin père de p_i, cette hauteur est notée $h(p_i)$. Enfin, la* hauteur *de T, notée H, est égale au maximum des hauteurs des processeurs dans T.*

Dans les définitions 5.5.3 et 5.5.4, nous distinguons l'arbre de parcours enraciné en *r* (*l'arbre normal*) des autres parcours (*les arbres anormaux*).

Définition 5.5.3 (Arbre normal) *Nous appelons* arbre normal *tout arbre contenant uniquement des processeurs p tels que $p =$ r $\vee \neg AbRoot(p)$.*

Remarque 5.5.2 *Trivialement, le système contient toujours un seul arbre normal : l'arbre enraciné en* r *(dans le cas où $S_r = C$, $Arbre($r$) = \emptyset$: $Arbre($r$)$ existe bien qu'il est vide).*

Définition 5.5.4 (Arbre anormal) *Tout arbre enraciné en un processeur différent de* r *est appelé* arbre anormal.

Dans les définitions 5.5.5 et 5.5.6, nous définissons la notion d'arbre *vivant* et d'arbre *mort*. Ces deux notions permettent de distinguer les parcours pouvant encore progresser (des parcours dont l'arbre associé est vivant) des parcours ne pouvant plus progresser (des parcours dont l'arbre associé est mort).

Définition 5.5.5 (Vivant) *Un arbre T satisfait le prédicat $Vivant(T)$ (ou est dit* vivant*) si et seulement si $\exists p \in T$ tel que $S_p \in Neig_p \cup \{P\}$.*

Définition 5.5.6 (Mort) *Un arbre T satisfait le prédicat $Mort(T)$ (ou est dit* mort*) si et seulement si $\neg Vivant(T)$.*

Remarque 5.5.3 *Aucun processeur ne peut s'accrocher à un arbre mort.*

Le concept d'arbre *E-mort* permet de représenter le gel des arbres anormaux.

Définition 5.5.7 (E-Mort) *Un arbre T satisfait le prédicat $E\text{-}Mort(T)$ (ou est dit* E-Mort*) si et seulement si $\forall p \in T$, $S_p \in \{EB,EF\}$.*

Remarque 5.5.4 $E\text{-}Mort(T) \Rightarrow Mort(T)$.

La notion de *S-Trace* permet de caractériser les états possibles d'un arbre de parcours.

Définition 5.5.8 (S-Trace) *Soit Y un tuple de processeurs ($Y = (p_0,p_1,\ldots,p_k)$). $S\text{-}Trace(Y) = S_0.S_1 \ldots S_k$ est la séquence des valeurs des variables S des processeurs p_i $(i = 0 \ldots k)$.*

Par définition, $S_r \in Neig_r \cup \{C,D,PC\}$ (cf. algorithme 5.5.5). Donc, si $S_r = C$, alors $Arbre(r) = \emptyset$ par la définition 5.5.2. Sinon ($S_r \in Neig_r \cup \{D,PC\}$), chaque fils, p, de r dans $Arbre(r)$ vérifie $(S_p \neq C) \wedge \neg AbRoot(p)$ avec $S_p \in Neig_p \cup \{C,P,D,PC,EB,EF\}$, d'après la définition 5.5.3 et l'algorithme 5.5.6. Ainsi $S\text{-}Trace(Par_p,p) \in \{p.p_i,p.P,q_i.D,PC.D,PC.PC\}$ où $Par_p = r$, $p_i \in Neig_p$ et $q_i \in Neig_r$ pour chaque fils p de r. D'où, par récurrence sur la hauteur des processeurs différents de r dans $Arbre(r)$, nous pouvons déduire le résultat suivant :

Lemme 5.5.1 *L'arbre normal, $Arbre(\mathrm{r})$, vérifie toujours l'un de ces deux cas :*

1. $Arbre(\mathrm{r}) = \emptyset \wedge S_r = C$, *ou*
2. $\forall p \in Arbre(\mathrm{r})$, $S\text{-}Trace((CheminPere(p) = p_0,p_1,\ldots,p_k)) \in q_0 \ldots q_j.P \cup q_0 \ldots q_j.D^* \cup PC^*.D^*$ *avec* $0 \leq j \leq k$ *et* $\forall i \in [0...j]$, $q_i \in Neig_{p_i}$.

Par définition, $\forall p \in V \setminus \{r\}$, $S_p \in Neig_p \cup \{C,P,D,PC,EB,EF\}$ (cf. algorithme 5.5.6). D'après les définitions 5.5.2 et 5.5.4, chaque fils q de p dans $Arbre(p)$ vérifie $q \neq r \wedge S_q \neq C \wedge \neg AbRoot(q)$ (dans chaque arbre anormale, seule la racine anormale vérifie $AbRoot$). Ainsi, $S\text{-}Trace(Par_q,q) \in \{q.P,p_i.D.,q.q_i,D.D,PC.D,PC.PC,EB.P,EB.D,EB.q_i,EB.PC,EB.EF,EF.EF\}$ où $Par_q = p$, $q_i \in Neig_q$ et $p_i \in Neig_p$ pour chaque fils q de p. D'où, par récurrence sur la hauteur des processeurs différents de p dans $Arbre(p)$, nous pouvons déduire le résultat suivant :

Lemme 5.5.2 *Soit T un arbre anormal. $\forall p \in T$, $S\text{-}Trace((CheminPere(p) = p_0,p_1,\ldots,p_k)) \in EB^*.q_i \ldots q_{j-1}.P \cup EB^*.q_i \ldots q_j.D^* \cup EB^*.PC^*.D^* \cup EB^*.EF^*$ avec $0 \leq i \leq j \leq k$ et $\forall t \in [i...j]$, $q_t \in Neig_{p_t}$.*

Preuve de la stabilisation instantanée en supposant un démon distribué faiblement équitable. Nous allons commencer par montrer que toute exécution du protocole $\mathcal{DFS}2$ est sans interblocage, *i.e.*, dans chaque configuration possible du système, il existe au moins un processeur activable (théorème 5.5.1). Pour cela, nous montrons tout d'abord que dans toute configuration contenant des arbres anormaux, il existe au moins un processeur activable (lemme 5.5.3).

Lemme 5.5.3 *$\forall \gamma \in \mathcal{C}$, si γ contient des arbres anormaux, alors $\exists p \in V$ tel que p est activable dans γ.*

Preuve. Supposons, par contradiction, qu'il existe une configuration γ qui contient des arbres anormaux mais dans laquelle aucun processeur n'est activable. Soit ar la racine anormale de γ ayant la valeur de L la plus grande. Soit T l'arbre anormal enraciné en ar.

1. Si $\exists p \in T$ tel que $S_p \notin \{EB,EF\}$, alors d'après le lemme 5.5.2 : soit $S_{ar} \notin \{EB,EF\}$ et la règle EB de ar est activable, contradiction ; soit, $\exists q \in CheminPere(p)$ tel que $S_q \notin \{EB,EF\} \wedge S_{Par_q} = EB$ et, dans ce cas, la règle EB de q est activable, contradiction.
2. D'où, par contradiction, $\forall p \in T$ $S_p \in \{EB,EF\}$ et, d'après le lemme 5.5.2, $S\text{-}Trace(CheminPere(p)) = EB^*.EF^*$.

(a) Si $S_{ar} = EF$, alors $\forall p \in T$, $S_p = EF$. De plus, $\forall q \in Neig_{ar}$ tel que $(Par_q = ar) \wedge (L_q > L_{ar})$, si $S_q \neq C$, alors $q \in T$ (sinon q vérifie $AbRoot(q) \wedge (L_q > L_{ar})$ et comme ar est une racine anormale dont la valeur de la variable L est la plus grande, on obtient une contradiction). Or, nous savons déjà que $\forall p \in T$, $S_p = EF$. Donc, $\forall q \in Neig_{ar}$ tel que $Par_q = ar$ et $L_q > L_{ar}$, nous avons $(S_q = C) \vee (S_q = EF)$ et la règle EC de ar est activable, contradiction.

(b) D'où, par contradiction, $S_{ar} = EB$ et $\exists p \in T$ tel que $(S_p = EB) \wedge (\forall q \in Child_p, S_q = EF)$ (en particulier, si p est une feuille de T, $Child_p = \emptyset$ et la condition est trivialement vérifiée). Or, similairement au cas précédent, nous pouvons conclure que dans ce cas la règle EF de p est activable, contradiction.

D'où, s'il existe au moins un arbre anormal dans γ, alors il existe au moins un processeur activable, contradiction. □

Théorème 5.5.1 *$\forall \gamma \in \mathcal{C}$, $\exists p \in V$ tel que p est activable dans γ.*

Preuve. Supposons, par contradiction, que $\exists \gamma \in \mathcal{C}$ telle que $\forall p \in V$, p n'est pas activable dans γ. D'après le lemme 5.5.3, cela signifie que γ ne contient pas d'arbres anormaux (en particulier, cela signifie que $\forall p \in V$, $S_p \notin \{EB,EF\}$ dans γ).

Supposons ensuite que $S_r = C$ dans γ. Alors $\forall p \in V$, $S_p = C$ (n.b. par hypothèse, il n'existe pas d'arbre anormal dans γ) et la règle B de *r* est activable, contradiction.

Supposons donc que $S_r \neq C$ dans γ :

1. Supposons alors que l'arbre normal, $Arbre(r)$, est mort dans γ. D'après la définition 5.5.6 et le lemme 5.5.1, $\forall p \in Arbre(r)$, $S\text{-}Trace(CheminPere(p)) = PC^+.D^*$. Maintenant, si $\exists p \in Arbre(r)$ tel que $S_p = D$, alors $\exists q \in CheminPere(p)$ ayant sa règle PC activable, contradiction. Donc, $\forall p \in Arbre(r)$, $S_p = PC$. Or, dans ce cas, la règle C de chaque feuille $Arbre(r)$ est activable, contradiction.
2. Supposons maintenant que l'arbre normal, $Arbre(r)$, est vivant dans γ. Pour obtenir une contradiction, nous allons tout d'abord montrer que s'il existe un processeur p tel que $S_p \neq C \wedge Que_p \neq A$, alors il existe au moins une règle de la phase de question qui est activable.
   - Supposons que $\exists p \in V$ tel que $(S_p \neq C) \wedge (Que_p = Q)$. Si $\exists q \in Neig_p$ tel que $(S_q \neq C) \wedge (Que_q \notin \{Q,R\})$, alors la règle R de q est activable, contradiction. Sinon $(\forall q \in Neig_p, (S_q \neq C) \Rightarrow (Que_q \in \{Q,R\}))$, la règle R de p est activable, contradiction. D'où, $\forall p \in V$, $(S_p \neq C) \Rightarrow (Que_p \neq Q)$.
   - Supposons maintenant que $\exists p \in V$ tel que $(S_p \neq C) \wedge (Que_p = R)$. Supposons ensuite que $p \neq r$. Si $Que_{Par_p} \neq R$, alors la règle R de Par_p est activable dans γ. Donc, pour chaque $p \neq r$ tel que $(S_p \neq C) \wedge (Que_p = R)$, nous avons $\forall q \in CheminPere(p)$, $Que_q = R$. Or, dans ce cas, au moins un des processeurs p, noté p', vérifie $\forall q \in Child_p, Que_q \in \{W,A\}$ (en particulier, si p est une feuille de l'arbre normal, nous avons $Child_p = \emptyset$ et la condition est trivialement satisfaite). Deux cas sont ensuite possibles :
     – p vérifie $\neg End(p)$ et sa règle W est activable, contradiction.
     – p vérifie $End(p)$. Si, en plus, $S_p = PC$, alors la règle C de p est activable, contradiction. Si $S_{S_p} = C$, alors la règle F de S_p est activable (S_p vérifie $Forward(S_p)$ car, comme, par hypothèse, il n'existe qu'un seul arbre : $Arbre(r)$, S_p vérifie $|Pred_{S_p}| = 1 \wedge End(S_p)$). Sinon, la règle W de p est activable, contradiction.

D'où, $\forall p \in V \backslash \{r\}$, $(S_p \neq C) \Rightarrow (Que_p \neq R)$ et, par contradiction, *r* est le seul processeur tel que $(S \neq C) \wedge (Que = R)$. Dans ce cas, nous savons déjà que $\forall p \in Child_r$, $Que_p \in \{W,A\}$ et $\forall q \in Neig_r$, $(S_q \neq C) \Rightarrow (Que_q \neq Q)$. Donc, la règle A de *r* est activable, contradiction. D'où, $\forall p \in V$, $(S_p \neq C) \Rightarrow (Que_p \neq R)$.

- Finalement, supposons qu'il existe $\exists p \in V$ tel que $(S_p \neq C) \wedge (Que_p = W)$. D'après les cas précédents, nous savons que $\forall q \in V$, $(S_q \neq C) \Rightarrow (Que_q \in \{W,A\})$ avec, en particulier, $S_r \neq C \wedge Que_r = A$ (par définition, $Que_r \neq W$). Or, dans une telle configuration, il est facile de montrer que tant qu'il existe des processeurs p tels que $(S_p \neq C) \wedge (Que_p = W)$, au moins un a sa règle A activable, contradiction.

D'où, par contradiction, $\forall p \in Arbre(r)$, $(S_p \neq C) \Rightarrow (Que_p = A)$ dans γ avec, en particulier, tout processeur p tel que $S_p \neq C$ vérifie $AnswerOk(p)$. Finalement, comme il existe exactement un arbre dans γ : l'arbre normal $Arbre(r)$ et que cet arbre est vivant, il existe exactement un processeur p qui vérifie l'une de ces conditions dans γ :

- $(p \neq r) \wedge (S_p = D) \wedge (S_{Par_p} = p)$. Or, nous savons déjà que Par_p vérifie $AnswerOk(Par_p)$ dans γ. Donc, Par_p vérifie $Backward(Par_p)$ et sa règle B est activable dans γ, contradiction.
- $S_p = P$. Dans ce cas, $p \neq r$ (l'état P n'existe pas pour *r*). Or, p vérifie $AnswerOk(p)$. Donc sa règle $Fbis$ est activable dans γ, contradiction.
- $S_p \in Neig_p \wedge End(p)$. Si $S_{S_p} = C$, S_p vérifie $Forward(S_p)$ car, comme, par hypothèse, il n'existe qu'un seul arbre : $Arbre(r)$, S_p vérifie $|Pred_{S_p}| = 1 \wedge End(S_p)$ et sa règle F est activable dans γ, contradiction. Si $S_{S_p} \neq C$ alors, comme il n'existe qu'un seul arbre : $Arbre(r)$ et que p vérifie $AnswerOk(p)$, p vérifie aussi $BadSucc(p)$ et sa règle IE est activable dans γ, contradiction.

□

La seconde étape de notre preuve (lemmes 5.5.4 à 5.5.8) consiste à montrer que le système ne contient plus de parcours anormaux en au plus $3n-3$ rondes. Pour cela, nous allons tout d'abord montrer que tous les arbres anormaux sont ***E-mort*** en au plus $n-1$ rondes (lemmes 5.5.4 à 5.5.7).

Lemme 5.5.4 *Lorsque la règle EB de p est activable, elle reste activable jusqu'à ce que p l'exécute.*

Preuve. Soit $\gamma \mapsto \gamma'$ un mouvement. Supposons, par contradiction, que la règle EB de p est activable dans γ et pas dans γ' (*i.e.*, $\neg EBroadcast(p)$ dans γ') bien que p n'exécute pas cette règle dans $\gamma \mapsto \gamma'$. Tout d'abord, comme la règle EB n'existe pas dans l'algorithme de *r*, nous savons que $p \neq r$ et $Par_p = q$ avec $q \in Neig_p$. Ensuite, $EBroadcast(p)$ est défini sur les variables S et L de p et q seulement. De plus, EB est la règle activable de p qui est la plus prioritaire (en effet, EC et EB ne peuvent pas être activables pour p dans la même configuration). Donc, p n'exécute pas de règle dans $\gamma \mapsto \gamma'$ et, par contradiction, q exécute une règle dans $\gamma \mapsto \gamma'$ qui modifie la valeur de S_q et/ou L_q pour que p vérifie $\neg EBroadcast(p)$ dans γ'. Ensuite, comme $S_p \notin \{C,EB,EF\}$ dans γ (car p vérifie $EBroadcast(p)$ dans γ) et que p n'exécute pas de règle dans $\gamma \mapsto \gamma'$, on a encore $S_p \notin \{C,EB,EF\}$ dans γ'. Enfin, par le prédicat $EBroadcast(p)$, nous savons que $[AbRoot(p) \vee (\neg AbRoot(p) \wedge S_{Par_p} = EB)]$ dans γ. D'où, l'étude des deux cas suivants :

- $AbRoot(p)$ dans γ. Comme $(AbRoot(p) \wedge S_p \notin \{C,EB,EF\}) \Rightarrow EBroadcast(p)$ et $S_p \notin \{C,EB,EF\}$ dans γ', on a $\neg AbRoot(p)$ dans γ'. Ensuite, $AbRoot(p) \equiv \neg GoodS(p) \vee \neg GoodL(p)$. Supposons alors que $\neg GoodL(p)$ dans γ. Comme $S_p \notin \{C,EB,EF\}$, $Par_p = q$, et $GoodL(p)$ dans γ' ($\neg AbRoot(p) \Rightarrow GoodL(p)$), $L_p = L_q + 1$ dans γ' et, pour cela, q doit exécuter sa règle F dans $\gamma \mapsto \gamma'$ (F est la seule règle qui modifie L_q). Or, comme $S_p \notin \{C,EB,EF\}$ et $Par_p = q$ dans γ, q vérifie $\neg Leaf(q)$ et la règle F de q n'est pas activable dans γ, contradiction. Donc, par contradiction, $GoodL(p) \wedge \neg GoodS(p)$ dans γ et $GoodS(p)$

dans γ'. Dans ce cas, $S\text{-}Trace(q,p) \in \{EF.P,\ PC.P,\ D.P,\ C.P,\ q_i.P,\ EF.p_i,\ PC.p_i,\ D.p_i,\ P.p_i.,\ C.p_i,\ q_i.p_i,\ C.D,\ P.D,\ EF.D,\ EF.PC,\ D.PC,P.PC,C.PC,q_j.PC\}$ dans γ où $Par_p = q$, $q_i \in Neig_q \setminus \{p\}$, $q_j \in Neig_q$ et $p_i \in Neig_p$.

1. Si $S\text{-}Trace(q,p) \in \{EF.P,C.P,EF.p_i,C.p_i,C.D,EF.D,EF.PC,C.PC\}$ dans γ alors, q ne peut exécuter aucune règle modifiant S_q dans $\gamma \mapsto \gamma'$ (d'après les algorithmes 5.5.5 et 5.5.6) donc $AbRoot(p)$ est toujours vérifié dans γ'. D'où la règle EB de p est toujours activable dans γ', contradiction.
2. Si $S\text{-}Trace(q,p) \in \{PC.P,D.P,PC.p_i,D.p_i,P.p_i,P.D,D.PC,P.PC\}$ dans γ, alors EB est la seule règle modifiant S_q qui peut être activable dans γ. Or, si q exécute EB dans $\gamma \mapsto \gamma'$, alors $S_q = EB$ dans γ' et, comme $S_p \notin \{C,EB,EF\}$ dans γ', la règle EB de p est toujours activable dans γ', contradiction.
3. Si $S\text{-}Trace(q,p) \in \{q_i.P,q_i.p_i,q_j.PC\}$ dans γ, alors les règles B, $Fbis$, IE et EB sont les seules règles modifiant S_q qui peuvent être activables dans γ. Or, si q exécute B, $Fbis$ ou IE dans $\gamma \mapsto \gamma'$, alors, $S\text{-}Trace(q,p) \in \{q_t.P,q_t.p_i,q_t.PC,D.P,D.p_i,D.PC\}$ avec $q_t \in Neig_q \setminus \{p\}$ et $AbRoot(p)$ est toujours vérifié dans γ'. D'où, la règle EB de p est toujours activable dans γ', contradiction. Enfin, si q exécute EB dans $\gamma \mapsto \gamma'$, alors, comme pour le cas précédent, la règle EB de p est toujours activable dans γ', contradiction.

- ($\neg AbRoot(p) \wedge S_{Par_p} = EB$) dans γ. En vérifiant chaque règle du protocole $\mathcal{DFS}2$, nous pouvons remarquer que, comme $S_q = EB$ ($Par_p = q$), EF est la seule règle que q peut exécuter $\gamma \mapsto \gamma'$. Or, ($\neg AbRoot(p) \Rightarrow GoodL(p)$) $\Rightarrow$ ($L_p = L_q + 1$) et ($L_p = L_q + 1 \wedge S_p \in \{B,F,P\}$) $\Rightarrow \neg EFeedback(q)$. Donc, la règle EF de q n'est pas activable dans γ et q n'exécute aucune règle dans $\gamma \mapsto \gamma'$. D'où, p est encore activable dans γ', contradiction.

Nous obtenons donc une contradiction dans tous les cas et le lemme est vérifié. □

Lemme 5.5.5 *Lorsque la règle EF de p est activable, elle reste activable jusqu'à ce que p l'exécute.*

Preuve. Soit $\gamma \mapsto \gamma'$ un mouvement. Supposons, par contradiction, que la règle EF de p est activable dans γ et pas dans γ' (*i.e.*, $\neg EFeedback(p)$ dans γ') bien que p n'exécute pas cette règle dans $\gamma \mapsto \gamma'$. Tout d'abord, comme la règle EF n'existe pas dans l'algorithme de *r*, nous savons que $p \neq r$ et $Par_p = q$ avec $q \in Neig_p$. Ensuite, comme EF est la règle de p qui est la plus prioritaire (quand la règle EF de p est activable, les règles EC et EB de p ne sont pas activables), p n'exécute pas de règle dans $\gamma \mapsto \gamma'$ et $S_p = EB$ dans γ'. Or, ($\neg EFeedback(p) \wedge S_p = EB$) $\Rightarrow$ ($\exists q \in Neig_p :: Par_q = p \wedge L_q > L_p \wedge S_q \notin \{EF,C\}$). Donc, il existe au moins un voisin, q, de p qui exécute une règle dans $\gamma \mapsto \gamma'$ pour satisfaire $Par_q = p \wedge L_q > L_p \wedge S_q \notin \{EF,C\}$ dans γ'. Deux cas sont alors possibles :

- q vérifie ($Par_q \neq p \vee L_q \leq L_p$) dans γ mais, après avoir exécuté une règle, q vérifie ($Par_q = p \wedge L_q > L_p \wedge S_q \notin \{EF,C\}$) dans γ' (en particulier, $Par_q = p$ dans γ' implique que $q \neq r$). Comme $Par_q = p \wedge L_q > L_p$ dans γ' et que F est la seule règle qui peut modifier les variables Par_q ou L_q, q exécute F dans $\gamma \mapsto \gamma'$. Or, par F, q peut seulement désigner comme père l'unique processeur q' tel que $S_{q'} = q$ dans γ (la règle F de q est activable seulement si $Pred_q = 1$). Or, $S_p = EB$ dans γ. Donc, $Par_q \neq p$ dans γ', contradiction.
- q vérifie ($Par_q = p \wedge L_q > L_p \wedge S_q \in \{EF,C\}$) dans γ mais, après avoir exécuté une règle, q vérifie ($Par_q = p \wedge L_q > L_p \wedge S_q \notin \{EF,C\}$) dans γ'. Si $S_q = C$ dans γ, alors q peut seulement exécuter sa règle F dans $\gamma \mapsto \gamma'$ et, comme dans le cas précédent, $Par_q \neq p$ dans γ', contradiction. Donc, $S_q = EF$ dans γ et la règle EC est la seule règle que q peut exécuter dans $\gamma \mapsto \gamma'$. Dans ce cas, q satisfait encore ($Par_q = p \wedge L_q > L_p \wedge S_q \in \{EF,C\}$) dans γ' (en effet, la règle EC exécute simplement $S_q := C$), contradiction.

Nous obtenons donc une contradiction dans tous les cas et le lemme est vérifié. □

Lemme 5.5.6 *Soit $p \in V$ tel que $S_p \in \{EB,EF\}$. p vérifie $S_p \in \{EB,EF\}$ jusqu'à ce que son arbre soit E-$mort$.*

Preuve. Soit $p \in V$ un processeur tel que $S_p \in \{EB,EF\}$. D'après les lemmes 5.5.1 et 5.5.2, p est dans un arbre anormal. Ensuite, $p \neq r$ (les états EB et EF n'existent pas pour *r*). D'après les règles de l'algorithme 5.5.6, nous pouvons alors affirmer que si p vérifie $S_p = EB$, alors p peut uniquement exécuter $S_p := EF$ par la règle EF. De la même manière, si p vérifie $S_p = EF$, alors p peut uniquement exécuter $S_p := C$ par la règle EC. De plus, p exécute EC seulement si $AbRoot(p)$. Donc, p exécute EC seulement s'il est une racine anormale avec $S_p = EF$. Or, d'après le lemme 5.5.2 si p est une racine anormale avec $S_p = EF$, alors son arbre $Arbre(p)$ est E-$mort$. □

En utilisant les lemmes précédents (lemmes 5.5.4 à 5.5.6), nous allons montrer que le gel atteint tous les processeurs des arbres anormaux, *i.e.*, tous les arbres anormaux sont ***E-mort***, en au plus $n-1$ rondes

Lemme 5.5.7 *Tous les arbres anormaux sont E-$mort$ en au plus $n-1$ rondes.*

Preuve. Soit $NotE_i$ l'ensemble des processeurs p tels que p est dans un arbre anormal et $S_p \notin \{EB,EF\}$ dans la première configuration de la $i^{\text{ème}}$ ronde. Soit $\mathcal{F} : \mathbb{N} \rightarrow \mathbb{N}$ la fonction définie comme suit :

$$\mathcal{F}(i) = \begin{cases} \infty & \textbf{si } NotE_i = \emptyset, \\ \min_{p \in NotE_i}(\{h(p)\}) & \textbf{sinon}. \end{cases}$$

D'après la définition 5.5.7, $\mathcal{F}(i) = \infty$ (*i.e.*, $NotE_i = \emptyset$) si et seulement si tous les arbres anormaux sont E-$mort$. Donc, pour prouver le lemme, nous devons montrer que $\mathcal{F}(n-1) = \infty$. Pour cela, nous allons tout d'abord montrer que tant que $NotE_i \neq \emptyset$, nous avons $\mathcal{F}(i) < \mathcal{F}(i+1)$.

Supposons, par contradiction, qu'il existe une exécution de $\mathcal{DFS}2$ dont la $i^{\text{ème}}$ ronde vérifie $\mathcal{F}(i) \geq \mathcal{F}(i+1)$ alors que $NotE_i \neq \emptyset$. Cela signifie qu'il existe au moins un processeur p au début de la $i+1^{\text{ème}}$ ronde qui vérifie $S_p \notin \{EB,EF\}$ et $h(p) \leq \mathcal{F}(i)$ (en particulier, p est dans un arbre anormal qui n'est pas E-$mort$). Or, d'après le lemme 5.5.6, nous savons que $\forall q \in V$ tel que $S_q \in \{EB,EF\}$, q vérifie $S_q \in \{EB,EF\}$ jusqu'à ce que son arbre soit E-$mort$. Deux cas sont donc possibles pour p :

a) Durant la $i^{\text{ème}}$ ronde, p s'accroche à un arbre anormal à la hauteur h telle que $h \leq \mathcal{F}(i)$ (n.b., p peut avoir quitté l'arbre auparavant). D'après les remarques 5.5.3 et 5.5.4, p s'accroche alors à un arbre qui n'est pas encore E-$mort$ en utilisant sa règle F. De plus, d'après la règle F, nous savons que p s'accroche à un arbre anormal à la hauteur h seulement s'il existe un voisin de p, q, tel que q est dans un arbre anormal, $S_q = p$ et $h(q) \leq \mathcal{F}(i) - 1$. Or, d'après la définition de $\mathcal{F}$ et le lemme 5.5.6, un tel processeur q n'existe pas, contradiction.

b) p vérifie $(S_p \notin \{C,EB,EF\}) \wedge (h(p) = \mathcal{F}(i))$ dans la première configuration de la $i^{\text{ème}}$ ronde, p reste dans cet arbre durant la $i^{\text{ème}}$ ronde et p vérifie encore $(S_p \notin \{C,EB,EF\}) \wedge (h(p) = \mathcal{F}(i))$ dans la première configuration de la $i+1^{\text{ème}}$ ronde.

   1. Si $h(p) = 0$, alors, d'après les définitions 5.5.2, 5.5.3 et 5.5.4, p vérifie $p \neq r$ et $AbRoot(p)$. D'après le prédicat $EBroadcast(p)$ et le lemme 5.5.4, la règle EB de p est alors continûment activable. Or, comme le démon est faiblement équitable, p affecte EB à S_p avant la fin de la ronde et, par le lemme 5.5.6, p vérifie alors $S_p \in \{EB,EF\}$ jusqu'à ce que son arbre soit E-$mort$, contradiction.

2. Si $h(p) > 0$, alors, d'après la définition 5.5.2, p vérifie $\neg AbRoot(p)$. Or, $(\neg AbRoot(p) \wedge S_p \notin \{EB,EF\}) \Rightarrow (S_{Par_p} \neq EF)$ et $(S_{Par_p} \neq EF \wedge h(Par_p) = \mathcal{F}(i)-1) \Rightarrow (S_{Par_p} = EB)$. Donc, d'après le prédicat $EBroadcast(p)$ et le lemme 5.5.4, la règle EB de p est alors continûment activable. Or, comme le démon est faiblement équitable, p affecte EB à S_p avant la fin de la ronde et, par le lemme 5.5.6, p vérifie alors $S_p \in \{EB,EF\}$ jusqu'à ce que son arbre soit $E\text{-}mort$, contradiction.

D'où, jusqu'à ce que $NotE_i = \emptyset$, nous avons $\mathcal{F}(i) < \mathcal{F}(i+1)$. Or, par définition, la valeur maximale de $\mathcal{F}(i)$ quand $NotE_i \neq \emptyset$ est égale à $n-2$, *i.e.*, la hauteur maximale d'un arbre anormal (tous les processeurs sauf *r* peuvent être dans un arbre anormal). Donc, dans le pire des cas, $\mathcal{F}$ est égale à ∞ pour $i = n-1$, *i.e.*, tous les arbres anormaux sont $E\text{-}mort$ en au plus $n-1$ rondes. □

Le lemme suivant montre que les arbres anormaux sont supprimés du système par la phase de correction en un nombre fini de rondes.

Lemme 5.5.8 *Le système ne contient plus d'arbres anormaux en au plus* $3n-3$ *rondes.*

Preuve. D'après les lemmes 5.5.2 et 5.5.7, en au plus $n-1$ rondes, chaque processeur p des arbres anormaux vérifie $S\text{-}Trace(CheminPere(p)) = EB^*.EF^*$ et, d'après les remarques 5.5.3 et 5.5.4, aucun processeur ne peut plus s'accrocher à ces arbres. Ensuite, nous pouvons remarquer qu'aucun processeur q ne peut quitter ces arbres anormaux avant de vérifier $S_p = EF$: seule une racine anormale peut quitter un arbre $E\text{-}mort$ via la règle EC lorsque tous les processeurs q de son arbre vérifient $S_q = EF$, d'après le lemme 5.5.2. De plus, tant qu'il existe des processeurs q dans des arbres anormaux tels que $S_q = EB$, il en existe au moins un, q', dont la règle EF est continûment activable, d'après les lemmes 5.5.2 et 5.5.5. Donc, le pire des cas est obtenu quand tous les processeurs des arbres anormaux vérifient $S = EB$. En effet, dans ce cas, il est nécessaire de propager la valeur EF à partir des feuilles des arbres jusq'à leur racine. $H+1$ rondes sont alors nécessaires pour cette propagation où H est la hauteur maximale d'un arbre anormal. Comme excepté *r*, tout processeur peut être dans un arbre anormal, cela implique que $H = n-2$. Ainsi, en au plus $n-1$ rondes supplémentaires, le système atteint une configuration γ où chaque processeur p dans un arbre anormal vérifie $S_p = EF$. Dans γ, $\forall p \in V$, p vérifie l'un de ces cas :

1. $S_p = C$,
2. $S_p = EF$ et p est dans un arbre anormal, ou
3. $S_p \in Neig_p \cup \{P,D,PC\}$ et p est dans l'arbre normal.

À partir de γ, les règles EC sont continûment activables pour chaque racine anormale jusqu'à ce que tous les arbres anormaux aient disparu. En effet, excepté les règles EC, les autres règles agissent uniquement sur l'arbre normal. Ainsi, les règles EC supprimeront les arbres anormaux à partir de leurs racines jusqu'aux feuilles en au plus $n-1$ rondes. D'où, le système ne contient plus d'arbre anormaux en au plus $3n-3$ rondes. □

Les lemmes suivants vont nous permettre de montrer qu'à partir de n'importe quelle configuration initiale, la racine initie, via sa règle F, un parcours en un nombre fini de rondes (théorème 5.5.2). Pour cela, nous allons tout d'abord montrer qu'à partir d'une configuration ne contenant pas d'arbres anormaux, la racine exécute sa règle F (l'action de démarrage) en un nombre fini de rondes (lemme 5.5.11). Cette preuve est décomposée en deux étapes :

1. Nous montrons que tout arbre normal dans une configuration ne contenant pas d'arbres anormaux est mort en un nombre fini de rondes (lemme 5.5.10).
2. Puis, nous montrons que la phase de nettoyage nettoie un tel arbre en un nombre fini de ronde pour permettre au système de démarrer à nouveau (lemme 5.5.11).

Lemme 5.5.9 *Tout processeur de l'arbre normal (*$Arbre(\mathrm{r})$*) peut le quitter seulement si l'arbre est mort.*

Preuve. D'après la définition 5.5.3, $\forall p \in Arbre(r)$, $p = r \vee \neg AbRoot(p)$. Donc, pour quitter $Arbre(r)$, un processeur doit exécuter la règle C et, par conséquent, doit être une feuille de l'arbre. Soit f une feuille de $Arbre(r)$. Pour quitter $Arbre(r)$, f doit vérifier $S_f = PC$. Or, d'après le lemme 5.5.1, si $S_f = PC$ alors $\forall p \in CheminPere(f)$, $S_p = PC$ avec, en particulier, $S_r = PC$. De plus, d'après la définition 5.5.3, *r* est l'extrémité initiale de $CheminPere(p)$, $\forall p \in Arbre(r) \setminus \{r\}$. Donc, $S_r = PC$ implique que $S_p \in \{D,PC\}$, $\forall p$, d'après le lemme 5.5.1, et $Arbre(r)$ est mort d'après la définition 5.5.6. □

Lemme 5.5.10 *À partir de toute configuration ne contenant pas d'arbres anormaux, l'arbre normal est mort en au plus* $4n^2 - 5n + 1$ *rondes.*

Preuve. Soit $\gamma \in \mathcal{C}$ une configuration ne contenant pas d'arbres anormaux. Nous savons qu'à partir de γ, l'exécution ne contient pas d'interblocage (théorème 5.5.1) et que, tant que l'arbre normal est vivant, chaque processeur peut s'y accrocher au plus une fois (lemme 5.5.9). Donc, à partir de γ, le pire des cas est le suivant : *r* vérifie $S_r \in Neig_r$ et tous les autres processeurs p vérifient $S_p = C$. En effet, dans ce cas, le système doit réaliser un parcours normal presque complet (un parcours complet sans la première règle : la règle F de *r*) avant que l'arbre normal soit mort.

Donc, d'après les algorithmes 5.5.5 et 5.5.6, à partir de γ, l'exécution se déroule comme suit. Tout d'abord, le jeton passe de voisin à voisin (règle F) jusqu'à un processeur p tel que $\forall q \in Neig_q$, $S_q \neq C$, *i.e.*, p est une feuille de l'arbre couvrant construit par $\mathcal{DFS}2$ (notez que après avoir exécuté sa règle F, un processeur a sa règle R immédiatement activable, donc les règles F et R sont exécutées en parallèle). Le processeur p est donc atteint en au plus H rondes où H est la hauteur maximum de l'arbre couvrant. En exécutant F, p affecte P à S_p. Une ronde supplémentaire est ensuite nécessaire pour que p affecte R à Que_p (règle R). Le processeur p propage alors une valeur W (règle W) dans le chemin de l'arbre le liant à la racine (pointeur Par). Après au plus H rondes, la valeur W atteint un fils de *r* et *r* vérifie $Answer(r)$. La racine déclenche alors une diffusion de la valeur A (règle A) dans son arbre. Après au plus $H + 1$ rondes, la valeur A atteint p, p vérifie alors $AnswerOk(p)$ et affecte D à S_p en exécutant sa règle $Fbis$. Cette dernière règle initie une remontée du jeton dans l'arbre (règle B), cette remontée s'arrête lorsque l'un de ces deux cas apparaît :

- Le jeton atteint un processeur q ayant des voisins n'appartenant pas à l'arbre. q désigne alors un nouveau successeur S_q et l'exécution de règles F reprend à partir de S_q, etc.
- Le jeton atteint *r* et *r* affecte D à S_r car tous ses voisins sont dans l'arbre. Dans ce cas, le lemme 5.5.1 implique que l'arbre normal est mort.

Cette phase de remontée est aussi effectuée en au plus $H + 1$ rondes.

Ainsi, jusqu'à ce que l'arbre normal $Arbre(r)$ soit mort, le parcours s'effectue en au plus $4H + 3$ rondes par feuille de l'arbre couvrant. Or, H est borné par $n - 1$ et, dans le pire des cas, l'arbre couvrant final est constitué de $n-1$ feuilles (tous les processeurs sauf *r*). D'où, à partir de γ, $Arbre(r)$ est mort en au plus $(n - 1) \times (4n - 1)$, *i.e.*, $4n^2 - 5n + 1$ rondes. □

Lemme 5.5.11 *À partir de n'importe quelle configuration ne contenant pas d'arbres anormaux,* r *exécute sa règle* F *en au plus* $4n^2 - 3n + 2$ *rondes.*

Preuve. D'après le lemme 5.5.10, à partir de n'importe quelle configuration ne contenant pas d'arbres anormaux, le système atteint une configuration γ où l'arbre normal est mort en au plus $4n^2 - 5n + 1$ rondes. Dans le pire des cas, γ correspond à la configuration suivante : $\forall p \in V$, $p \in Arbre(r)$ $\wedge$ $S_p = D$. En effet, dans ce cas, le système doit exécuter une phase de nettoyage complète avant

de pouvoir à nouveau démarrer : la valeur PC doit être diffusée dans l'arbre à partir de *r* jusqu'aux feuilles (règle PC). Cette diffusion coûte au plus n rondes. Ensuite, l'arbre se nettoie à partir des feuilles grâce à l'exécution de règles C en au plus n rondes. D'où, après au plus $4n^2 - 3n + 1$ rondes, le système atteint une configuration où $\forall p \in V$, $S_p = C$. Dans une telle configuration, *r* est le seul processeur activable et donc exécute la règle F dans le mouvement suivant (resp. la ronde suivante). □

D'après les lemmes 5.5.8 et 5.5.11, nous pouvons affirmer le résultat suivant :

Théorème 5.5.2 *À partir de n'importe quelle configuration,* r *exécute sa règle F en au plus $4n^2 - 1$ rondes.*

Nous allons maintenant montrer que tout parcours initié par *r* (règle F) est un parcours en profondeur du réseau (théorème 5.5.3). Ainsi nous pourrons conclure que le protocole $\mathcal{DFS}2$ est instantanément stabilisant (théorème 5.5.4). Le lemme suivant montre un résultat technique utilisé dans la preuve du théorème 5.5.3.

Lemme 5.5.12 *Soit p un processeur d'un arbre anormal tel que $Que_p \in \{Q,R\}$. Tant que p ne quitte pas l'arbre, p vérifie $Que_p \neq A$.*

Preuve. Soit p un processeur d'un arbre anormal tel que $Que_p \in \{Q,R\}$. Supposons que p vérifie aussi $AbRoot(p)$. Si $S_p \in \{EB,EF\}$, alors d'après le lemme 5.5.6, p vérifie $S_p \in \{EB,EF\}$ jusqu'à ce qu'il quitte l'arbre (cf. prédicat $Answer(p)$). Donc, la règle A de p n'est pas activable jusqu'à ce qu'il quitte l'arbre. Si $S_p \in Neig_p \cup \{P,D,PC\}$, alors sa règle EB est continûment activable (lemme 5.5.4). Or, la règle EB est plus prioritaire que la règle A. Donc, p ne peut pas exécuter A avant EB et, si p exécute EB, alors on retrouve le cas précédent.

Supposons maintenant que p vérifie $\neg AbRoot(p)$. Deux cas sont alors possibles :

- $Que_p = R$. La valeur R est alors propagée dans $CheminPere(p)$: pour tout processeur q dans $CheminPere(p)$, dès que $Que_q = R$, q ne pourra modifier Que_q que via la règle W et seulement quand Que_{Par_q} sera égal à R. Ensuite, pour affecter A à Que_q, q devra vérifier, en particulier, $Que_q = W$ et $Que_{Par_q} = A$ (seule *r* peut passer de R à A). Donc, nous pouvons remarquer qu'il existera toujours une valeur R dans $CheminPere(p)$ pour empêcher q de passer de W à A : les valeurs R sont une barrière entre les valeurs A et W. D'où, tant que p est dans l'arbre, p ne peut pas affecter A à Que_p.
- $Que_p = Q$. Si p reste dans l'arbre, alors Que_p reste égal à Q jusqu'à ce que p exécute sa règle R. Or, si p exécute sa règle R, alors on retrouve le cas précédent.

D'où, tant que p ne quitte pas l'arbre, p vérifie $Que_p \neq A$. □

Avant de démontrer que tout parcours initié par *r* est un parcours en profondeur (théorème 5.5.3), nous allons montrer un résultat intermédiaire : tout parcours initié par *r* visite tous les processeurs du réseau (lemme 5.5.13).

Lemme 5.5.13 *À partir de n'importe quelle configuration où* r *exécute sa règle F, $\mathcal{DFS}2$ parcourt tous les processeurs du réseau.*

Preuve. Supposons, par contradiction, qu'il existe certains processeurs qui ne seront pas visités lors du parcours initié par *r*. Alors, comme le réseau est connexe, il existe au moins un processeur $p \neq r$ qui ne sera pas visité tandis qu'un de ses voisins q le sera. Le processeur q exécute la règle F pour recevoir le jeton du parcours normal pour la première fois (*i.e.*, q s'accroche au parcours initié par *r*). En exécutant F, nous pouvons alors remarquer que $Que_q := Q$ et que q ne peut pas affecter directement D à S_q (cf. macro $Next_q$).

- Supposons alors que $S_p \neq C$. Si $Que_p \in \{W,A\}$, alors S_q reste différent de D et Que_q reste égal à Q jusqu'à ce que $Que_p = R$. Ainsi, p finit par exécuter la règle R et p vérifie ensuite $Que_p \in \{Q,R\}$. Or, comme $S_p \neq C$ et p n'est jamais visité, p est dans un arbre anormal et, d'après le lemme 5.5.12, $Que_p \neq A$ tant que p ne quitte pas son arbre. Donc, tant que p ne quitte pas son arbre, q ne peut pas affecter D à S_q via les règles $Fbis$ ou B. En effet, q ne vérifie pas $AnswerOk(q)$ à cause de p. Ainsi, d'après le lemme 5.5.8, p finit par quitter son arbre et le système atteint une configuration où $S_p = C$.
- Supposons maintenant que p vérifie (ou finisse par vérifier) $S_p = C$. Si p s'accroche (une nouvelle fois) à un arbre anormal, p exécute en particulier $Que_p := Q$ (cf. règle F) et, comme précédemment, tant que p ne quitte pas l'arbre, q ne peut pas affecter D à S_q. Or, d'après le lemme 5.5.8, le système finit par ne plus contenir d'arbre anormal. Donc, p finit par vérifier $S_p = C$ pour toujours. De plus, tant que $S_p = C$, q ne peut pas affecter D à S_q (cf. macro $Next_q$). Or, d'après le lemme 5.5.10, l'arbre normal finit par être mort. Donc, q finit par affecter D à S_q et nous obtenons une contradiction.

D'où, p finit par être visité par le parcours normal et cela signifie qu'à partir d'une configuration où *r* exécute sa règle F, tous les processeurs sont visités en un temps fini par le parcours initié par *r*. □

Théorème 5.5.3 *À partir de n'importe quelle configuration où* r *exécute sa règle* F*,* $\mathcal{DFS}2$ *exécute un parcours en profondeur du réseau.*

Preuve. D'après le lemme 5.5.13, nous savons qu'à partir d'une configuration où *r* exécute sa règle F, le parcours initié par *r* visite séquentiellement tous les processeurs du réseau. Pour montrer que le parcours effectué est un parcours en profondeur, nous allons montrer qu'il suit la méthode présentée dans la remarque 5.3.1 : il progresse le plus profondément possible dans le réseau avant de remonter dans l'arbre du parcours. Pour cela, nous allons prouver que chaque processeur p visité par le parcours issu de *r* affecte D à S_p seulement quand tous ses voisins appartiennent à l'arbre normal : $Arbre(r)$ (*i.e.* quand tous ses voisins ont été visités par le parcours courant). Cette preuve est fournie ci-dessous.

Quand p est visité pour la première fois par le parcours initié par *r* (règle F), il affecte en particulier Q à Que_p. Ensuite, Que_p reste égal à Q tant que $\exists q \in Neig_p$ tels que $S_q \neq C \wedge Que_q \neq R$. Donc, chaque processeur q finit par exécuter $Que_q := R$ (règle R). De plus, chaque fois qu'un voisin de p s'accroche à un arbre anormal (par la règle F), il exécute en particulier $Que := Q$. Donc, d'après le lemme 5.5.12, p vérifiera $(S_p \neq C \wedge Que_p = A) \wedge (\forall q \in Neig_p, Que_q = A \wedge S_q \neq C)$ seulement quand $\forall q \in Neig_p$, $q \in Arbre(p)$. D'où, p affectera D à S_p seulement quand tous ses voisins auront été visités par le parcours initié par *r* (cf. prédicat $AnswerOk(p)$ et macro $Next_p$ dans les règles $Fbis$ et B). D'où, le parcours séquentiel initié par *r* est un parcours en profondeur.

Nous pouvons donc conclure qu'à partir de n'importe quelle configuration où *r* exécute sa règle F, $\mathcal{DFS}2$ exécute un parcours en profondeur du réseau. □

D'après la remarque 3.2.1, les théorèmes 5.5.2 et 5.5.3, nous pouvons déduire le théorème suivant :

Théorème 5.5.4 $\mathcal{DFS}2$ *est un protocole instantanément stabilisant de parcours en profondeur sous un démon distribué faiblement équitable.*

Preuve de la stabilisation instantanée en supposant un démon distribué inéquitable. Nous avons montré que $\mathcal{DFS}2$ est un protocole instantanément stabilisant de parcours en profondeur sous un démon faiblement équitable (théorème 5.5.4). Donc, pour prouver la stabilisation instantanée du protocole sous l'hypothèse d'un démon inéquitable, il reste à montrer que chaque parcours exécuté par notre protocole est réalisé en un nombre fini de mouvements (cf. plan de la preuve présenté dans le premier paragraphe de la sous-section 5.5.2).

Nous commençons cette preuve avec les deux lemmes techniques suivants. Ces lemmes seront utilisés plus tard dans la preuve.

Lemme 5.5.14 $\forall p \in V \setminus \{\mathrm{r}\}$ *tel que* $AbRoot(p)$, p *ne peut pas exécuter de règle de la phase de question.*

Preuve. Supposons, par contradiction, que p exécute une règle de la phase de question dans $\gamma \mapsto \gamma'$. Alors, d'après l'algorithme 5.5.6, p vérifie, en particulier, $S_p \notin \{C,EB,EF\}$ dans γ. Or, $(S_p \notin \{C,EB,EF\} \wedge AbRoot(p)) \Rightarrow EBroadcast(p)$ et, comme la règle EB est plus prioritaire que toute règle de la phase de question, p peut seulement exécuter la règle EB dans $\gamma \mapsto \gamma'$, contradiction. □

Lemme 5.5.15 *L'exécution de chaque règle* F *engendre l'exécution de* $O(\Delta \times n)$ *règles de la phase de question.*

Preuve. Suite à l'exécution de sa règle F, un processeur p vérifie $S_p \in Neig_p \cup \{P\}$, $Par_p \in Neig_p$ et $Que_p = Q$. Ensuite, Que_p reste égal à Q jusqu'à ce que $\forall q \in Neig_p$, $(S_q \neq C) \Rightarrow (Que_q \in \{Q,R\})$. Si un voisin de p, noté q, exécute $Que_q := R$ via sa règle R (dans ce cas, $S_q \neq C$), alors q vérifie $Que_q = R$ jusqu'à ce que p exécute $Que_p := R$ (règle R). Enfin, quand chaque voisin q de p vérifie $(S_q \neq C) \Rightarrow (Que_q \in \{Q,R\})$, p peut exécuter $Que_p := R$ (règle R). Donc, chaque exécution de la règle F engendre au plus $\Delta + 1$ nouvelles valeurs R dans le système.

Ensuite, toutes ses valeurs R ($O(\Delta)$) sont propagées (par l'exécution de règles R) uniquement dans les $CheminPeres$ de p et de chaque q vérifiant $S_q \neq C$ jusqu'à un processeur v tel que $(v = r) \vee AbRoot(Par_v)$ (cf. prédicat $Require$ et lemme 5.5.14). Or, le nombre de processeurs qui composent chaque $CheminPere$ est borné par n. Donc, le coût de ces propagations est en $O(\Delta \times n)$ règles.

Puis, dans le pire des cas, p et chacun de ses voisins q tel que $S_q \neq C$ ($O(\Delta)$) exécute $Que := W$ via la règle W. Dans le pire des cas encore, les valeurs W sont propagées (par l'exécution de règles W) uniquement dans les $CheminPeres$ de p et de chaque q vérifiant $S_q \neq C$ jusqu'à un processeur v tel que $(v = r) \vee AbRoot(Par_v)$ (cf. prédicat $Wait$ et lemme 5.5.14). Donc, similairement aux valeurs R, le coût de ces propagations est en $O(\Delta \times n)$ règles.

Finalement, dans le pire des cas, les $O(\Delta)$ valeurs W créées précédemment sont toutes propagées dans l'arbre normal. Dans ce cas, chaque valeur W finit par atteindre un fils de r et peut provoquer la diffusion d'une valeur A dans tout l'arbre normal (r est le seul processeur capable de générer une valeur A). Donc, le coût de ces diffusions est encore en $O(\Delta \times n)$ règles (n.b. l'arbre normal peut contenir tous les processeurs du réseau).

D'où, chaque règle F engendre l'exécution de $O(\Delta \times n)$ règles de la phase de question. □

Nous allons maintenant montrer (lemmes 5.5.16 à 5.5.21) que les arbres anormaux ne peuvent engendrer l'exécution que d'un nombre fini de règles. Pour cela, nous étudions le nombre d'actions des phases de visite, de nettoyage et de correction engendrées par les arbres anormaux (le nombre d'actions de la phase de question a déjà été traité dans un cadre général, cf. lemme 5.5.15).

Tout d'abord, nous montrons que les arbres anormaux n'engendrent l'exécution que d'un nombre fini de règles de la phase de visite (lemme 5.5.16 à lemme 5.5.18).

Lemme 5.5.16 $\forall p \in V \setminus \{\mathrm{r}\}$, *après s'être accroché à un arbre anormal (règle* F*),* p *ne peut quitter son arbre avant que celui-ci ne soit mort.*

Preuve. Supposons, par contradiction, qu'après s'être accroché à un arbre anormal, un processeur p quitte son arbre alors qu'il est toujours vivant. Deux règles de p lui permettent de quitter un arbre (*i.e.*, $S_p := C$) : les règles C et EC.

– Supposons que p quitte son arbre via la règle C. Pour exécuter la règle C, p doit vérifier, en particulier, $S_p = PC$ (cf. prédicat $Clean(p)$). Or, après s'être accroché via la règle F, p vérifie $S_p \in Neig_p \cup \{P\}$ et $Que_p = Q$. En particulier, $Que_p \neq A$ jusqu'à ce que p quitte son arbre d'après le lemme 5.5.12. Ensuite, d'après l'algorithme 5.5.6, p doit exécuter $S_p := D$ via la règle $Fbis$, B, ou IE, puis $S_p := PC$ via la règle PC pour finalement vérifier $S_p = PC$. Or, pour exécuter $S_p := D$ via la règle $Fbis$, B, ou IE, p doit vérifier, en particulier, $Que_p = A$ (cf. $AnswerOk(p)$), contradiction.
– Supposons que p quitte son arbre via la règle EC. Pour exécuter la règle EC, p doit vérifier, en particulier, $(S_p = EF) \wedge AbRoot(p)$ (cf. prédicat $EFAbRoot(p)$). Donc, d'après la définition 5.5.2, p est la racine anormale de son arbre. De plus, comme $S_p = EF$, le lemme 5.5.2 implique que $\forall q \in Arbre(p)$, $S_q = EF$, *i.e.*, $Arbre(p)$ est mort d'après la définition 5.5.6. D'où, p peut quitter son arbre via la règle EC seulement quand son arbre est mort, contradiction.

□

D'après le lemme 5.5.16 et la remarque 5.5.3, nous déduisons le lemme suivant :

Lemme 5.5.17 *Dans une exécution, $\forall p,q \in V \setminus \{\mathrm{r}\}$, q s'accroche à l'arbre anormal enraciné en p (règle F) au plus une fois.*

Lemme 5.5.18 *Soit p un processeur dans un arbre anormal. p exécute au plus une règle de la phase de visite avant de quitter son arbre.*

Preuve. Supposons que p (un processeur dans un arbre anormal) exécute une action de la phase de visite dans $\gamma \mapsto \gamma'$. D'après les définitions 5.5.2, 5.5.3 et 5.5.4, $p \neq r$ et $S_p \neq C$ dans γ. Donc, p peut exécuter la règle $Fbis$, B, ou IE dans $\gamma \mapsto \gamma'$ et soit $S_p = D$ soit $S_p = q$ avec $q \in Neig_p$ dans γ'.

Si $S_p = D$ dans γ', alors, d'après l'algorithme 5.5.6, p ne peut plus exécuter de règle de la phase de visite jusqu'à ce qu'il quitte son arbre.

Si $S_p = q$ ($q \in Neig_p$) dans γ', alors $S_q = C$ dans γ. Ensuite, d'après l'algorithme 5.5.6, la règle B est la seule règle de la phase de visite que p pourra exécuter s'il reste dans son arbre. La règle B de p sera activable seulement quand q vérifiera $(Par_q = p) \wedge (S_q = D)$. Or, comme $S_q = C$ dans γ, q doit s'accrocher à p (règle F) puis p doit exécuter $S_q := D$. Après s'être accroché via la règle F, q vérifie $S_q \in Neig_p \cup \{P\}$ et $Que_q = Q$. En particulier, $Que_q \neq A$ jusqu'à ce que q quitte son arbre d'après le lemme 5.5.12. Or, d'après l'algorithme 5.5.6, q peut exécuter $S_p := D$ seulement via les règles $Fbis$, B ou IE et pour exécuter de telles règles q doit vérifier, en particulier, $Que_q = A$ (cf. $AnswerOk(q)$), contradiction. Donc, p ne peut plus exécuter de règle de la phase de visite tant qu'il est dans cet arbre (à cause de q).

D'où, p exécute au plus une règle de la phase de visite avant de quitter son arbre. □

Nous montrons maintenant que les arbres anormaux n'engendrent l'exécution que d'un nombre fini de règles de la phase de nettoyage (lemme 5.5.19).

Lemme 5.5.19 *Dans une exécution, un processeur p quitte un arbre anormal en exécutant sa phase de nettoyage (au plus deux règles) seulement s'il est dans cet arbre depuis la configuration initiale.*

Preuve. Supposons, par contradiction, que p quitte un arbre anormal en exécutant sa phase de nettoyage après s'y être accroché (règle F). En exécutant la règle F, p affecte q à S_p tel que $q \in Neig_p \cup \{P\}$ et Q à Que_p. Donc, $Que_p \neq A$ jusqu'à ce que p quitte l'arbre d'après le lemme 5.5.12. D'après l'algorithme 5.5.6, cela signifie que D ne peut pas être affecté à S_p tant que p ne quitte pas son arbre. D'où, d'après l'algorithme 5.5.6, p ne pourra pas quitter son arbre anormal en exécutant sa phase de nettoyage, contradiction. □

Le lemme suivant montre que les arbres anormaux n'engendrent l'exécution que d'un nombre fini de règles de la phase de correction (lemme 5.5.20).

Lemme 5.5.20 *Soit p un processeur dans un arbre anormal. p quitte son arbre après avoir exécuté au plus trois règles de la phase de correction.*

Preuve. Ce lemme se déduit trivialement d'après les règles de l'algorithme 5.5.6. □

Grâce aux lemmes 5.5.15 à 5.5.20, nous pouvons maintenant calculer le nombre total d'actions engendrées par les arbres anormaux durant l'exécution :

Lemme 5.5.21 *Les arbres anormaux engendrent l'exécution de $O(\Delta \times n^3)$ règles avant de disparaître.*

Preuve. D'après les définitions 5.5.1 et 5.5.4, au plus $n-1$ processeurs sont dans des arbres anormaux dans la configuration initiale du système. De plus, chaque processeur excepté *r* ($O(n)$) peut s'accrocher à chaque arbre anormal au plus une fois durant l'exécution (lemme 5.5.17). Ensuite, chaque processeur qui s'accroche à un arbre anormal engendre l'exécution de $O(\Delta \times n)$ règles de la phase de question (lemme 5.5.15) et exécute au plus quatre autres règles avant de quitter son arbre (lemmes 5.5.18 à 5.5.20). Naturellement, le nombre d'exécution de règles engendrées par un processeur dans un arbre anormal dès la configuration initiale est du même ordre. Donc, chaque arbre anormal engendre l'exécution de $O(\Delta \times n^2)$ règles avant de disparaître et comme, par la définition 5.5.4, il existe au plus $n-1$ arbres anormaux dans la configuration initiale, le lemme est vérifié. □

Nous nous focalisons maintenant sur l'arbre normal. Nous allons montrer que l'arbre normal n'engendre l'exécution que d'un nombre fini de règles avant de vérifier $S_r = C$ (lemmes 5.5.22 à 5.5.24). Pour cela, nous allons étudier le nombre d'actions des phases de visite et de nettoyage engendrées par l'arbre normal avant que $S_r = C$ (le nombre d'actions de la phase de question a déjà été traité dans un cadre général, cf. lemme 5.5.15).

Tout d'abord, nous nous focalisons sur la phase de visite.

Lemme 5.5.22 *L'arbre normal ($Arbre(\mathrm{r})$) est mort après avoir engendré l'exécution de $O(n)$ règles de la phase de visite.*

Preuve. L'exécution d'une règle de la phase de visite est engendrée par l'arbre normal dans le mouvement $\gamma \mapsto \gamma'$ pour effectuer l'une de ces tâches :

- désigner un nouveau successeur au parcours de *r*, *i.e.* une variable S_p reçoit q tel que $q \in Neig_p$ où p est soit un processeur de l'arbre normal dans γ (Règles $Fbis$, B ou IE) ou un processeur s'accrochant à l'arbre normal dans $\gamma \mapsto \gamma'$ (règle F).
- affecter P à une variable S_p où p est un processeur s'accrochant à l'arbre normal dans $\gamma \mapsto \gamma'$ (règle F).
- affecter D à une variable S_p où p est un processeur de l'arbre normal dans γ (Règles $Fbis$, B ou IE).

Donc, pour connaître le nombre d'exécutions de règles de la phase de visite engendrées par l'arbre normal avant qu'il ne meure, il suffit de compter le nombre total de modifications subit par les variables S_p de tous les processeurs p initialement dans l'arbre normal ou s'accrochant à l'arbre normal avant qu'il ne meure.

Tout d'abord, nous comptons le nombre d'exécutions de règles permettant de désigner les successeurs lors du parcours de *r*. Supposons que p s'accroche (par la règle F) à $Arbre(r)$ dans $\gamma \mapsto \gamma'$. Pour s'accrocher à $Arbre(r)$, p a été précédemment désigné comme successeur par un de ses voisins q tel que $q \in Arbre(r)$. Ensuite, après avoir désigné p, q n'a plus de règles de la phase de visite activable jusqu'à ce que p devienne son fils en s'accrochant à $Arbre(r)$. Donc, il y a autant d'exécutions de règles pour désigner les successeurs du parcours de *r* que de processeurs s'accrochant à $Arbre(r)$.

Or, $O(n)$ processeurs s'accrochent à $Arbre(r)$ jusqu'à ce que $Arbre(r)$ ne meure (cf. lemme 5.5.9). D'où, jusqu'à ce qu'il soit mort, $Arbre(r)$ engendre l'exécution de $O(n)$ règles pour désigner les successeurs du parcours de *r*.

Nous comptons maintenant le nombre d'exécutions de règles permettant d'affecter P aux variables S des processeurs s'accrochant à $Arbre(r)$. Un processeur $p \neq r$ peut affecter P à S_p uniquement en exécutant la règle F : lorsqu'il s'accroche à $Arbre(r)$. Donc le nombre d'exécutions de ses règles est aussi en $O(n)$ (lemme 5.5.9).

Finalement, nous comptons le nombre d'exécutions de règles permettant d'affecter D aux variables S des processeurs de $Arbre(r)$. Quand p affecte D à S_p (Règles $Fbis$, B ou IE) cela signifie qu'il considère que tous ses voisins ont été visités par le parcours de *r*. S_p reste ensuite égal à D et p n'exécute plus de règles de la phase de visite jusqu'à ce qu'il quitte $Arbre(r)$ d'après l'algorithme 5.5.6, *i.e.*, jusqu'à ce que $Arbre(r)$ soit mort d'après le lemme 5.5.9. Donc, le nombre d'exécutions de ses règles est aussi en $O(n)$.

D'où, l'arbre normal ($Arbre(r)$) est mort après avoir engendré l'exécution de $O(n)$ règles de la phase de visite. □

Nous regardons maintenant la phase de nettoyage.

Lemme 5.5.23 *Si l'arbre normal est mort, alors il sera complètement nettoyé (*i.e., $S_r = C$*) après avoir engendré l'exécution de $O(n)$ règles de la phase de nettoyage.*

Preuve. Si $Arbre(r)$ est mort, alors il vérifie : $\forall p \in Arbre(r)$, $S\text{-}Trace((CheminPere(p)) \in PC^*.D^*$ d'après le lemme 5.5.1. Donc, d'après les algorithmes 5.5.5 et 5.5.6, seules les règles de nettoyage peuvent s'appliquer sur cet arbre jusqu'à ce que $S_r = C$. La phase de nettoyage est composée de deux règles : PC et C. La règle PC propage la valeur PC de *r* jusqu'aux feuilles de $Arbre(r)$. Ensuite, l'arbre est nettoyé des feuilles jusqu'à *r* par l'exécution de règles C. D'où, après l'exécution de $O(n)$ règles de la phase de nettoyage sur l'arbre (au plus deux règles par processeurs), le système atteint une configuration où $S_r = C$ et le lemme est vérifié. □

Grâce aux lemmes 5.5.15, 5.5.22 et 5.5.23, nous pouvons maintenant conclure sur le nombre total d'actions engendrées par l'arbre normal avant que $S_r = C$.

Lemme 5.5.24 *L'arbre normal engendre l'exécution de $O(\Delta \times n^2)$ règles avant de vérifier $S_r = C$.*

Preuve. Tout d'abord, d'après la définition 5.5.3, l'arbre normal contient uniquement des processeurs p tels que $(p = r) \vee \neg AbRoot(p)$. Donc, aucun processeur de l'arbre n'exécute de règles de la phase de correction. Ensuite, jusqu'à ce qu'il soit mort, l'arbre normal progresse dans le réseau via les phases de visite et de question. La phase de visite est réalisée grâce à l'exécution de $O(n)$ de ses règles (lemme 5.5.22). De plus, durant cette phase, comme $O(n)$ règles F sont générées, $O(\Delta \times n^2)$ règles de la phase de question sont aussi générées (lemme 5.5.15). Naturellement, le coût de la phase de question (en terme de règles) pour les processeurs présents dans l'arbre normal depuis la configuration initiale est du même ordre. Finalement, la phase de visite termine lorsque l'arbre normal est mort. Le système est alors dans une configuration où $\forall p \in Arbre(r)$, $S\text{-}Trace((CheminPere(p)) \in PC^*.D^*$ (lemme 5.5.1). Chaque processeur de l'arbre n'exécute plus d'action de la phase de visite jusqu'à ce qu'il quitte l'arbre et l'arbre est nettoyé après l'exécution de $O(n)$ règles de la phase de nettoyage (lemme 5.5.23). □

D'après le lemme 5.5.21, nous savons que les arbres anormaux engendrent l'exécution d'un nombre fini de règles, *i.e.*, $O(\Delta \times n^3)$ exécutions de règles, avant de disparaître. Donc, le démon inéquitable ne peut empêcher de manière permanente la progression de l'arbre normal. Or, l'arbre normal engendre l'exécution de $O(\Delta \times n^2)$ règles avant de vérifier $S_r = C$ (lemme 5.5.24). Donc, le système atteint

une configuration où $S_r = C$ après $O(\Delta \times n^3)$ mouvements. À partir de cette configuration et jusqu'à ce que *r* exécute sa règle F, toutes les règles exécutées dans le système sont engendrées par les arbres anormaux. Une nouvelle fois, d'après le lemme 5.5.21, après $O(\Delta \times n^3)$ mouvements supplémentaires, *r* exécute sa règle F. D'où, le théorème suivant est vérifié.

Théorème 5.5.5 *À partir de n'importe quelle configuration initiale,* r *exécute sa règle F en $O(\Delta \times n^3)$ mouvements.*

Corollaire 5.5.1 *À partir de n'importe quelle configuration initiale, le protocole $\mathcal{DFS}2$ exécute un parcours en profondeur (complet) en $O(\Delta \times n^3)$ mouvements.*

D'après le théorème 5.5.4 et le corollaire 5.5.1, nous avons :

Théorème 5.5.6 *$\mathcal{DFS}2$ est un protocole instantanément stabilisant de parcours en profondeur sous un démon distribué inéquitable.*

5.5.3 Complexité

Complexité en espace. Dans les algorithmes 5.5.5 et 5.5.6, les valeurs de la variable L ne sont pas bornées. Cependant, nous pouvons remarquer que le protocole *$\mathcal{DFS}2$* reste valide si nous bornons L par n. Donc, chaque processeur a besoin de $\log n$ bits pour stocker sa variable L et en tenant compte des autres variables déclarées dans les algorithmes 5.5.5 et 5.5.6, nous pouvons déduire le théorème suivant :

Théorème 5.5.7 *L'occupation mémoire de $\mathcal{DFS}2$ est en $O(\log n)$ bits par processeur.*

Par rapport à la solution précédente (*$\mathcal{DFS}1$*), le protocole *$\mathcal{DFS}2$* a un surcoût modéré. En effet, comme nous savons qu'un protocole de parcours en profondeur non tolérant aux fautes a besoin de $\Theta(\Delta)$ bits par processeur, le surcoût en mémoire de *$\mathcal{DFS}2$* est en $O(\log n / \log \Delta)$.

Complexité en temps. Nous analysons maintenant la complexité en temps du protocole *$\mathcal{DFS}2$* en fonction des critères présentés dans la sous-section 3.2.3.

Temps de stabilisation. D'après le théorème 5.5.6, *$\mathcal{DFS}2$* est instantanément stabilisant. Donc, par la définition 3.2.1, le temps de stabilisation de *$\mathcal{DFS}2$* est zéro à la fois en nombre de rondes et en nombre de mouvements.

Délai. D'après les théorèmes 5.5.2 et 5.5.5, nous savons que le délai du protocole *$\mathcal{DFS}2$* est en $O(n^2)$ rondes et $O(\Delta \times n^3)$ mouvements, respectivement.

Le nombre de fois où le protocole doit être initié pour exécuter un parcours conforme aux spécifications. Par définition, pour n'importe quel protocole instantanément stabilisant, ce nombre est égal à un. Or, d'après le théorème 5.5.6, *$\mathcal{DFS}2$* est instantanément stabilisant.

Le nombre de fois où un processeur peut participer à un parcours corrompu. D'après le lemme 5.5.17, chaque processeur peut s'accrocher au plus une fois à chaque arbre anormal. Or, initialement, le système contient au plus $n - 1$ arbres anormaux. Donc, durant l'exécution un processeur peut participer à $O(n)$ parcours corrompus.

Le nombre de fois où un processeur peut participer à un parcours corrompu sans le détecter comme tel. D'après le lemme 5.5.19, lorsqu'un processeur p s'accroche à arbre enraciné en un processeur q

tel que $q \neq r$, il ne peut le quitter qu'en exécutant sa phase de correction et, dans ce cas, p détecte qu'il a participé à un parcours corrompu. Donc, $\forall p \in V$, p peut participer à un parcours enraciné en un processeur q tel que $q \neq r$ sans le détecter qu'au plus une fois : si p participe à ce parcours en continu depuis la configuration initiale. Ainsi, p peut participer au plus deux fois à un parcours corrompu sans le détecter :

- Un parcours enraciné en r mais pas initié par r.
- Un parcours enraciné en q tel que $q \in V \setminus \{r\}$.

Le temps d'exécution nécessaire pour effectuer le premier parcours conforme aux spécifications. D'après le théorème 5.5.6, nous savons qu'à partir de n'importe quelle configuration initiale, le premier parcours initié par la racine sera conforme aux spécifications. Or, d'après le corollaire 5.5.1, nous savons qu'à partir de n'importe quelle configuration initiale, le protocole $\mathcal{DFS}2$ exécute un parcours en profondeur (complet) en $O(\Delta \times n^3)$ mouvements. Le théorème suivant nous donne le temps d'exécution du premier parcours (conforme aux spécifications) en nombre de rondes. Il se déduit des lemmes 5.5.8, 5.5.11 et du théorème 5.5.2.

Théorème 5.5.8 *À partir de n'importe quelle configuration initiale, le protocole $\mathcal{DFS}2$ exécute un parcours en profondeur (complet) en au plus $8n^2 - 3n + 1$ rondes.*

Le temps d'exécution nécessaire pour effectuer les parcours suivants. Comme nous l'avons affirmé dans la sous-section 3.2.3, de manière générale, pour un protocole stabilisant (auto-stabilisation ou stabilisation instantanée), le pire des cas du temps d'exécution d'un cycle (complet) de calcul conforme aux spécifications correspond au pire des cas de l'exécution du premier cycle conforme aux spécifications. C'est le cas aussi avec le protocole $\mathcal{DFS}2$. En effet, le temps d'exécution d'un parcours passe de $O(\Delta \times n^3)$ (corollaire 5.5.1) à $O(n^2)$ mouvements (théorème 5.5.9). Cela est dû au fait que le premier parcours initié par r force la destruction des arbres anormaux de sorte que le deuxième parcours initié démarre à partir d'une configuration initiale normale (lemme 5.5.25).

Lemme 5.5.25 *Après le premier parcours initié par* r, *les autres parcours démarrent toujours d'une configuration telle que $\forall p \in V$, $S_p = C$.*

Preuve. D'après le théorème 5.5.3 et le lemme 5.5.9, nous pouvons déduire que lorsque le premier parcours initié termine à r, le système est dans une configuration où tous les processeurs sont dans l'arbre normal. Donc à partir de cette configuration, le système ne contient plus d'arbres anormaux et suite au nettoyage de l'arbre normal, le système atteint une configuration initiale normale, *i.e.* une configuration telle que $\forall p \in V$, $S_p = C$. □

Théorème 5.5.9 *Après le premier parcours initié par* r, *les autres parcours sont exécutés en $O(n^2)$ mouvements.*

Preuve. D'après le lemme 5.5.25, suite au premier parcours initié par r, les autres parcours démarrent toujours d'une configuration γ telle que $\forall p \in V$, $S_p = C$. À partir de γ, le système ne contient plus jamais d'arbres anormaux. Donc, aucune règle de la phase de correction ne sera plus exécutée durant le reste de l'exécution. Ensuite, $O(n)$ règles de la phase de visite (lemme 5.5.22) et $O(n)$ règles de la phase de nettoyage (lemme 5.5.23) sont exécutées durant le parcours. Considérons maintenant la phase de question. Soit $p \in V$.

1. Durant un parcours à partir de γ, p exécute la règle R une fois pour passer de Q à R après s'être accroché à l'arbre normal. Ensuite, p exécutera la règle R chaque fois qu'il devra propager vers r une valeur R initiée par l'un de ses descendants dans l'arbre couvrant, *i.e.*, $O(n)$ règles R.

2. De la même manière, p exécutera la règle W chaque fois qu'il devra propager vers *r* une valeur W initiée par une feuille (de l'arbre couvrant) appartenant à son sous-arbre, *i.e.*, $O(n)$ règles W.
3. Finalement, p exécutera la règle A chaque fois qu'il devra propager une valeur A initiée par *r* à la suite d'une requête (W) initiée par une feuille (de l'arbre couvrant) appartenant à son sous-arbre, *i.e.*, $O(n)$ règles A.

D'où, chaque processeur exécute $O(n)$ règles de la phase de question durant un parcours initié à partir de γ : $O(n^2)$ règles de la phase de question et le théorème est vérifié. □

Le surcoût en temps. Pour évaluer le surcoût en temps de notre protocole, nous avons besoin de connaître :

(1) D'une part, le temps d'exécution d'un parcours à partir d'une configuration initiale normale. Cette complexité est en $O(n^2)$ rondes (lemme 5.5.11) et $O(n^2)$ mouvements (lemme 5.5.25 et théorème 5.5.9).
(2) D'autre part, le temps d'exécution d'un parcours effectué dans un système sans faute par le protocole le plus efficace. Or, nous avons vu dans la section 5.3 que cette complexité est identique en terme de mouvements et de rondes : $2(n-1)$.

En tenant compte de ces deux complexités, nous pouvons déduire que le surcoût en temps de la stabilisation de notre protocole (*i.e.* le rapport en (1) et (2)) est de l'ordre de n à la fois en rondes et en mouvements.

5.5.4 Conclusion

Dans cette section, nous avons présenté un second protocole de parcours en profondeur instantanément stabilisant pour un réseau enraciné quelconque fonctionnant avec un démon distribué inéquitable. Comme pour la première solution proposée dans la section 5.4, ce protocole n'utilise pas de structures sous-jacentes telles que, par exemple, un arbre couvrant et n'a pas besoin de connaissances globales sur le réseau telles que sa taille, par exemple. De plus, comme précédemment, la propriété de stabilisation instantanée assure que dès que la racine initie un nouveau parcours, tous les processeurs sont visités dans un ordre induit par la profondeur d'abord. Cependant, ce protocole est moins efficace que le précédent en temps d'exécution : par exemple, le délai est en $O(n^2)$ rondes et $O(\Delta \times n^3)$ mouvements alors que nous obtenons pour la même tâche $O(n)$ rondes et $O(n^2)$ mouvements avec la solution de la section 5.4. Le temps d'exécution des parcours (comme pour $\mathcal{DFS}1$, le premier parcours est plus coûteux que les autres) et le surcoût en temps sont aussi moins efficaces. En revanche, cette nouvelle solution apporte des améliorations majeures. Premièrement, le protocole $\mathcal{DFS}2$ n'utilise plus les identités des processeurs et son occupation mémoire est en $O(\log n)$ bits par processeur au lieu de $O(n \times \log n)$ pour la solution précédente. Ensuite, $\mathcal{DFS}2$ détecte efficacement les parcours corrompus : un processeur peut participer au plus deux fois à un parcours corrompu sans le détecter (cf. sous-section 5.5.3). Enfin, nous pouvons remarquer qu'à partir du deuxième démarrage, notre protocole réalise un parcours en profondeur premier (comme défini par Collin et Dolev dans [CD94]). En effet, d'après le lemme 5.5.25, suite au premier parcours initié par la racine, le système démarre toujours à partir d'une configuration telle que $\forall p \in V$, $S_p = C$ (une configuration initiale normale). À partir d'une telle configuration, nous pouvons remarquer que le système contient toujours au plus un jeton et que, $\forall p \in V$, lorsque p détient le jeton, il désigne toujours comme successeur son voisin non-visité minimal dans l'ordre local $\prec_p$ (si un tel processeur existe). D'où, à partir de la deuxième exécution de $\mathcal{DFS}2$, l'arbre couvrant construit est un point fixe.

5.6 Gestion explicite des requêtes

Les protocoles $\mathcal{DFS}1$ et $\mathcal{DFS}2$ font explicitement référence aux requêtes, via la variable $Request_r$, dans le code de l'algorithme de leur initiateur *r* (cf. algorithmes 5.4.3 et 5.5.5). Cependant, dans un premier temps, nous n'avons pas tenu compte de la variable $Request_r$ dans l'analyse de ces protocoles (*i.e.*, la preuve de la stabilisation instantanée et l'analyse de complexité) car, d'une part, la prise en compte de la variable $Request_r$ alourdirait inutilement les preuves et d'autre part la gestion de la variable $Request_r$ est identique pour les deux protocoles. En préambule, nous allons rappeler comment la variable $Request_r$ intervient dans les deux protocoles (sous-section 5.6.1). Puis, nous montrerons que la manière dont nous gérons les requêtes n'altère pas la validité de nos protocoles (sous-section 5.6.2). Enfin, nous nous intéresserons aux répercussions sur la complexité des protocoles (sous-section 5.6.3).

5.6.1 Mise en oeuvre de la requête

Dans les protocoles $\mathcal{DFS}1$ et $\mathcal{DFS}2$, la variable $Request_r$ apparaît uniquement au niveau de l'action de démarrage : la règle F de *r* et de l'action de terminaison : la règle T de *r*. $Request_r$ apparaît à la fois dans la garde et dans le traitement de ces deux règles. Ainsi, la garde de la règle F est de la forme suivante : $Forward(r) \wedge (Request_r = Wait)$. Cela signifie que l'initiateur *r* peut exécuter une action de démarrage seulement si (1) son état ainsi que celui de ses voisins le permettent ($Forward(r)$) et (2) une demande a été effectuée via la règle externe IR ($Request_r = Wait$). Ensuite, en cas d'exécution de la règle F, la présence de l'affectation $Request_r := In$ dans le traitement de la règle permet de signifier à l'application (ou l'utilisateur) demandeuse que le cycle de calcul demandé est démarré, *i.e.*, la requête est prise en compte. Pour la règle T, nous avons une garde de la forme $Forward(r) \wedge (Request_r = In)$ et le traitement associé suivant : $Request_r := Out$. En effet, après que le cycle de calcul (en l'occurrence le parcours) initié se termine à la racine par l'exécution de la règle C, *r* finit par vérifier à nouveau $Forward(r)$. Dans ce cas, l'initiateur informe l'application (ou l'utilisateur) demandeur que le cycle de calcul demandé est terminé et qu'il est à nouveau possible d'effectuer une requête (cf. règle IR page 36) en exécutant la règle T. Nous pouvons alors remarquer que *r* n'est plus activable jusqu'à ce qu'une nouvelle requête soit effectué : jusqu'à l'exécution de la règle IR.

5.6.2 Validité de la gestion explicite des requêtes

Nous allons maintenant montrer que la manière dont nous gérons les requêtes n'altère pas la validité de nos protocoles.

Théorème 5.6.1 *$\mathcal{DFS}1$ (resp. $\mathcal{DFS}2$) reste instantanément stabilisant lorsque les requêtes sont explicitement gérées.*

Preuve. La gestion explicite de la requête ne modifie pas le comportement des variables internes des protocoles $\mathcal{DFS}1$ et $\mathcal{DFS}2$. En fait, la variable $Request_r$ peut juste bloquer l'action de démarrage lorsque $Request_r \neq Wait$, *i.e.* en l'absence de demande (cf. algorithmes 5.4.3 et 5.5.5). Donc, pour montrer que la gestion explicite de la requête n'altère pas la validité de nos protocoles, il suffit de montrer qu'elle ne crée pas d'interblocage, *i.e.*, aucune règle interne du protocole ne sera plus jamais activable seulement si plus aucune demande n'est effectuée par l'application.

Les seules règles dépendant de $Request_r$ sont les règles F et T. Or, nous avons démontré que, quelle que soit la configuration initiale, sans la gestion explicite des requêtes, *r* pouvait appliquer la règle F en un temps fini. La modification dûe à la gestion de la variable $Request_r$ nous amène

à dire que *r* peut appliquer les règles internes Fou T si $Request_r \neq Out$. Si *r* peut appliquer T alors la valeur de $Request_r$ passe à Out et l'interface est déclenchable. Donc, si l'application est demandeuse, $Request_r$ passe à $Wait$ et pour la même raison que *r* pouvait exécuter T, *r* peut alors exécuter F. D'où, aucune règle interne au protocole n'est plus jamais activable seulement si la règle externe IR n'est plus jamais activable bien que $Request_r = Out$. Or, d'après la garde de la règle IR cela signifie que le prédicat $ApplicationRequest(r)$ n'est plus jamais satisfait, c'est à dire, plus aucune demande ne sera jamais effectuée par l'application. □

5.6.3 Conséquences sur la complexité en temps

Nous allons maintenant montrer que la gestion explicite des requêtes engendre un surcoût en temps négligeable.

Théorème 5.6.2 *Avec une gestion explicite des requêtes, le délai pour exécuter une action de démarrage demandée de $\mathcal{DFS}1$ (resp. $\mathcal{DFS}2$) est du même ordre que le délai calculé sans tenir compte de cette gestion explicite.*

Preuve. Supposons que l'application (ou l'utilisateur) demande un cycle de calcul (*i.e*, le prédicat $ApplicationRequest(r)$ est vérifié).

- Si $Request_r = Out$, alors la règle IR devient alors immédiatement activable. L'exécution de cette règle fait passer $Request_r$ à $Wait$ et la garde de la règle F dépend maintenant de $Forward(r)$ uniquement. Donc, le délai pour exécuter l'action de démarrage demandé de $\mathcal{DFS}1$ (resp. $\mathcal{DFS}2$) est égal au délai précédemment calculé plus une action.
- Si $Request_r = In$, il faut que $Request_r$ passe de In à Out (règle T) puis de Out à $Wait$ (règle IR). Donc, par rapport au délai précédemment calculé, nous avons un surcoût de deux actions. D'où, le délai total pour exécuter l'action de démarrage demandé de $\mathcal{DFS}1$ (resp. $\mathcal{DFS}2$) est égal au délai calculé sans tenir compte de la gestion explicite des requêtes plus deux actions.
- Enfin, le cas où $Request_r = Wait$ représente le pire des cas. En effet, il faut que le système exécute un cycle complet de calcul non demandé avant que la règle IR puis l'action de démarrage ne soient exécutées, c'est à dire, le temps d'exécution d'un cycle complet précédemment calculé plus l'exécution des règles T et IR. Cependant, nous avons vu dans les sous-sections 5.4.3 et 5.5.3 que le délai et le temps d'exécution d'un cycle de calcul complet de $\mathcal{DFS}1$ (resp. $\mathcal{DFS}2$) sans tenir compte de la gestion explicite des requêtes sont du même ordre. D'où, dans ce cas aussi, le délai total de $\mathcal{DFS}1$ (resp. $\mathcal{DFS}2$) reste du même ordre que le délai calculé sans tenir compte de la gestion explicite des requêtes.

□

Suivant le même argumentaire, nous pouvons montrer que lorsque l'application (ou l'utilisateur) demande un cycle de calcul de $\mathcal{DFS}1$ (resp. $\mathcal{DFS}2$), le temps d'exécution de ce cycle de calcul est du même ordre que celui calculé sans tenir compte de la gestion explicite des requêtes. D'où le corollaire suivant :

Corollaire 5.6.1 *Avec une gestion explicite des requêtes, lorsque l'application (ou l'utilisateur) demande un cycle de calcul de $\mathcal{DFS}1$ (resp. $\mathcal{DFS}2$), le temps d'exécution de ce cycle de calcul est du même ordre que celui calculé sans tenir compte de cette gestion explicite.*

5.7 Applications fondées sur le parcours en profondeur instantanément stabilisant

Le parcours en profondeur permet d'évaluer des propriétés globales sur le réseau. Dans cette section, nous donnons deux exemples de calculs de propriétés globales utilisant le parcours en profondeur. La première application (cf. sous-section 5.7.3) est un calcul de point fixe avec détection de terminaison. L'algorithme présenté permet de marquer les *points d'articulation* (définition A.2.15, page 161) et les *isthmes* (définition A.2.16, page 161) du réseau. Les points d'articulation (resp. les isthmes) sont des processeurs (resp. des canaux) dont la suppression (suite à une panne définitive, par exemple) provoque la partition du réseau en plusieurs composantes connexes (cf. définitions A.2.15 et A.2.16, page 161, pour les définitions formelles). De plus, l'existence de points d'articulation ou d'isthmes dans le réseau peut être l'une des causes d'apparition de congestions. L'identification des points d'articulation et des isthmes est donc essentielle du point de vue de la tolérance aux fautes. La seconde application (cf. sous-section 5.7.4) permet d'évaluer si un ensemble fourni par l'application est un *ensemble séparateur* (définition A.2.14, page 161) du réseau. Un ensemble séparateur (*cutset*) est un sous-ensemble de processeurs du réseau dont la suppression (suite à des pannes définitives, par exemple) provoque la partition du réseau en plusieurs composantes connexes (cf. définition A.2.14, page 161, pour la définition formelle). La détection d'ensembles séparateurs est un problème important dans de nombreuses applications telles que l'évaluation de la fiabilité des réseaux. Nos deux applications sont obtenues par la composition conditionnelle (cf. sous-section 5.7.2) entre un algorithme distribué calculant la propriété recherchée et un des deux protocoles de parcours en profondeur instantanément stabilisants précédemment présentés. En fait, nos deux applications utilisent deux propriétés communes des protocoles $\mathcal{DFS}1$ et $\mathcal{DFS}2$ (cf. sous-section 5.7.1). Dans les preuves de ces deux protocoles, nous allons utiliser de nombreux termes empruntés à la théorie des graphes. Les définitions de ces termes sont fournies en annexe (section A, page 159). Pour ces définitions, nous nous sommes basés sur les livres de Berge [Ber83] et d'Aho et Hullman [AU92].

5.7.1 Rappel sur les protocoles DFS1 et DFS2

Pour réaliser nos applications, nous pouvons utiliser l'un ou l'autre des deux protocoles de parcours en profondeur instantanément stabilisants précédemment présentés. En fait, nos deux applications reposent sur deux propriétés communes des protocoles $\mathcal{DFS}1$ et $\mathcal{DFS}2$. Ces propriétés sont les suivantes :

Propriété 5.7.1 *Dans les protocoles $\mathcal{DFS}1$ et $\mathcal{DFS}2$, $\forall p \in V$, lorsque p participe à un parcours initié par* r, *p affecte D à S_p uniquement quand tous ses voisins ont été visités par ce parcours.*

Propriété 5.7.2 *Dans les protocoles $\mathcal{DFS}1$ et $\mathcal{DFS}2$, $\forall p \in V$, à partir du moment où p participe à un parcours initié, p ne participera plus jamais à un parcours corrompu.*

La première propriété est dûe au fait que les protocoles $\mathcal{DFS}1$ et $\mathcal{DFS}2$ suivent le schéma d'algorithme présenté dans la remarque 5.3.1. Pour le protocole $\mathcal{DFS}1$, la deuxième propriété est montrée dans la preuve du théorème 5.4.11 (page 65). Pour le protocole $\mathcal{DFS}2$, la deuxième propriété se déduit du théorème 5.5.3 (page 88) et du lemme 5.5.9 (page 86).

5.7.2 Composition conditionnelle

Pour écrire nos applications, nous allons utiliser une technique de composition de protocoles : la *composition conditionnelle*. Cette technique de composition a été introduite par Datta *et al* dans [DGPV00]. Elle permet notamment de simplifier l'écriture et les preuves des protocoles.

Définition 5.7.1 (Composition Conditionnelle) *Soit P_1 et P_2 deux protocoles tels que toute variable écrite par P_2 n'apparaît pas dans le code de P_1. La* composition conditionnelle *de P_1 et P_2, notée $P_2 \circ_{|\mathcal{G}} P_1$, est un protocole qui vérifie les conditions suivantes :*

1. *$P_2 \circ_{|\mathcal{G}} P_1$ contient toutes les variables et règles de P_1 et P_2.*
2. *$\mathcal{G}$ est un sous-ensemble de gardes de P_1.*
3. *Chaque garde de P_2 est de la forme $g \wedge h$ ou $\neg g \wedge h$ où g est expression booléenne utilisant les gardes de $\mathcal{G}$.*
4. *Comme une règle de P_2 peut être activable quand une règle de P_1 est activable, l'ordre de l'exécution est le suivant : exécution de la règle de P_2 suivit par l'exécution de la règle de P_1 dans le même mouvement.*

La définition suivante sera utilisée pour prouver la stabilisation instantanée de nos protocoles composites.

Définition 5.7.2 (Exécution Équitable [Tel01]) *Une exécution e de la composition P_1 et P_2 est équitable pour P_i avec $P_i \in \{1, 2\}$ si et seulement si l'une de ses conditions est vérifiée :*

- *e est finie.*
- *e contient une infinité d'exécution de règles de P_i ou contient un suffixe infini dans lequel aucune règle de P_i n'est activable.*

5.7.3 Points d'articulation et isthmes

À l'origine, le problème de l'identification des points d'articulation et des isthmes a été abordé dans la théorie des graphes : Paton [Pat71] puis Tarjan [Tar72] ont proposé des solutions pour les graphes de topologie quelconque (n.b. le protocole de Tarjan a une complexité linéaire). Ce problème a ensuite été abordé dans le contexte de l'algorithmique distribuée non tolérante aux fautes : [AZ89, PTMH91, SK92]. Enfin, Karaata et Chaudhuri ont proposé des solutions auto-stabilisantes dans [Kar99, KC99, Cha99b, Cha99a]. Toutes ces solutions (*i.e.*, [Kar99, KC99, Cha99b, Cha99a]) supposent l'existence d'un arbre couvrant le réseau : les solutions proposées dans [Cha99b, Cha99a] utilisent un arbre couvrant en profondeur (pouvant être calculé par l'un des protocoles silencieux de [Her91a, CD94, AB98, DT01]) tandis que dans [Kar99, KC99], un arbre couvrant en largeur est requis (cet arbre pouvant être calculé, en particulier, par le protocole silencieux de Huang et Chen, [HC92]). Les protocoles de [Cha99b, Cha99a] ont la meilleure complexité : à partir de n'importe quelle configuration, le système atteint un état terminal où tous les points d'articulation et isthmes du réseau sont identifiés en $O(n^2)$ mouvements au lieu de $O(n^2 \times m)$ mouvements pour [Kar99, KC99] (sans tenir compte du temps de calcul de l'arbre couvrant). Cependant, ces solutions ont deux inconvénients majeurs. Le premier concerne leur occupation mémoire : $\Omega(m \times \log(m))$ bits par processeur (en effet, chaque processeur maintient des listes d'arêtes). Le second inconvénient est que ces protocoles sont silencieux donc, par définition, aucun processeur du réseau n'est capable de détecter quand tous les points d'articulation et isthmes du réseau sont identifiés (*i.e.*, quand le point fixe est atteint). Enfin, il faut noter que nous avons proposé un protocole de détection des points d'articulation et des isthmes dans [Dev05]. Ce protocole suppose aussi l'existence d'un arbre couvrant en profondeur dans le réseau. Ce protocole améliore significativement l'occupation mémoire des solutions précédentes : $O(\log \Delta + \log n)$ bits par processeur, tout en gardant une complexité en temps en $O(n^2)$ mouvements (sans tenir compte du temps de calcul de l'arbre couvrant). Néanmoins, cette solution est encore une fois un protocole silencieux.

Nous allons maintenant proposer un protocole instantanément stabilisant permettant d'identifier les points d'articulation et les isthmes du réseau. Une propriété intéressante de ce protocole est

qu'il termine à la racine après que le point fixe est atteint, *i.e.*, seulement après que tous les points d'articulation et les isthmes du réseau sont localement identifiés (marqués). Ce protocole est en fait la composition entre un algorithme distribué calculant une fonction $u(p)$ pour chaque processeur $p \neq r$ et un algorithme instantanément stabilisant de parcours en profondeur. La fonction $u(p)$ permet d'identifier si p est un point d'articulation et si l'arête entre p et son père dans l'arbre couvrant est un isthme. Notre approche est basée sur deux résultats de Tarjan. Ces résultats utilisent les propriétés intrinsèques des arbres couvrants en profondeur. Soit $Arbre(r) = (V,E_T)$ un arbre couvrant en profondeur du réseau $G = (V,E)$. Soit $Fils(p)$ l'ensemble des fils de p dans $Arbre(r)$, $\forall p \in V$. Soit $E \setminus E_T$ l'ensemble des *arêtes hors-arbre*. Les deux théorèmes suivants ont été établis par Tarjan dans [Tar72] :

Théorème 5.7.1 r *est un point d'articulation de* G *si et seulement si* $|Fils(\mathrm{r})| \geq 2$.

Théorème 5.7.2 $\forall p \in V \setminus \{\mathrm{r}\}$, p *est un point d'articulation si et seulement si* $\exists q \in Fils(p)$ *tel qu'aucun processeur parmi* q *et ses descendants dans* $Arbre(\mathrm{r})$ *n'est lié par une* arête hors-arbre *à un ancêtre de* p.

Les deux théorèmes de Tarjan sont basés sur une propriété intrinsèque des arbres couvrants en profondeur. Cette propriété est donnée ci-dessous (elle est facilement déductible de la définition des arbres couvrants en profondeur, définition A.2.12, page 161).

Propriété 5.7.3 $\forall p \in V$, *chaque descendant* q *de* p *dans un arbre couvrant en profondeur,* $Arbre(\mathrm{r})$, *vérifie : chaque voisin hors-arbre de* q *est soit un ancêtre soit un descendant de* p *dans* $Arbre(\mathrm{r})$.

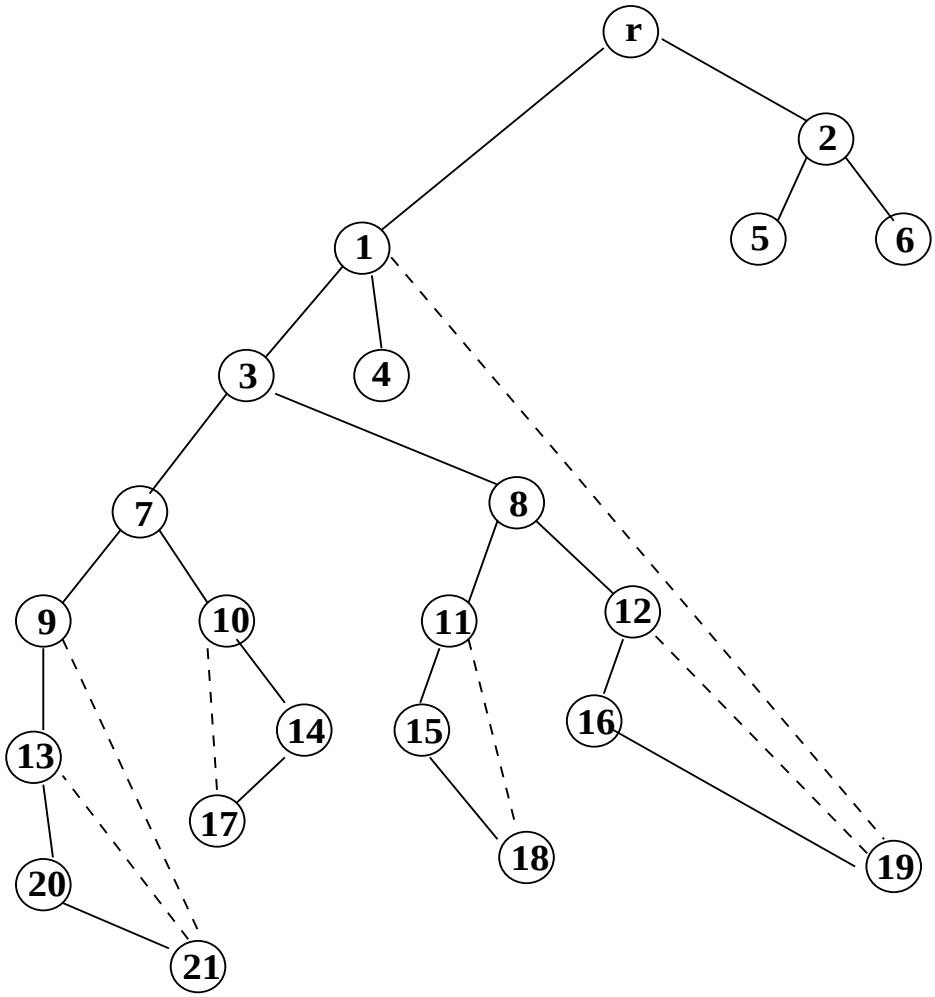
La remarque suivante montre que les points d'articulation et les isthmes sont deux notions proches.

Remarque 5.7.1 *Une arête est un isthme si et seulement si chacun de ses sommets incidents est un sommet pendant (*i.e.*, un sommet de degré un) ou un point d'articulation.*

La figure 5.7.15 illustre les deux théorèmes de Tarjan et la remarque 5.7.1. Dans cette figure, un arbre en profondeur du graphe enraciné en *r* est décrit par les arêtes en trait plein. La racine à deux fils dans cet arbre. D'après le Théorème 5.7.1, la racine est donc un point d'articulation. En effet, si nous supprimons le sommet *r* et ses arêtes incidentes, nous obtenons un graphe non-connexe avec deux composantes : le sous-graphe induit par 1 et ses descendants et le sous-graphe induit par 2 et ses descendants. Le processeur 3 est lui aussi un point d'articulation d'après le Théorème 5.7.2. En effet, aucun des descendants de son fils 7 (*i.e.*, 9, 10, 13, 14, 17, 20, 21) n'est relié à un ancêtre de 3 (*i.e.*, 1 ou *r*) par une arête hors-arbre. D'ailleurs, si nous supprimons 3 et ses arêtes incidentes alors le sous-graphe induit par 7 et ses descendants se retrouve déconnecté du reste du graphe. Finalement, nous pouvons remarquer que, par exemple, l'arête $\{2,5\}$ est un isthme. En effet, 2 est un point d'articulation et 5 un sommet pendant (cf. remarque 5.7.1).

Approche. En nous basant sur les résultats précédents (*i.e.*, les théorèmes 5.7.1, 5.7.2 et la remarque 5.7.1), nous introduisons la fonction $u(p)$, $\forall p \in V \setminus \{r\}$. La fonction $u(p)$ utilise la notion de *voisin hors-arbre, i.e.*, les voisins liés à un processeur par une arête hors-arbre (n.b. d'après la propriété 5.7.3, tout voisin hors-arbre q d'un processeur p est soit un ancestre soit un descendant de p donc, nous pouvons avoir $q \in Arbre(p)$). Nous définissons $u(p)$ comme suit :

$$\forall p \in V \setminus \{r\},\ u(p) = \min_{x \in Arbre(p)} (\{h(y) :: \{x,y\} \in E \setminus E_T\} \cup \{h(x)\}).$$



arête de l'arbre couvrant

arête hors−arbre

r, 1, 2, 3, 7, 8, 9, 10, 11 sont des points d'articulation

(r,1), (r,2), (1,3), (1,4), (2,5), (2,6), (3,7),
(7,9), (7,10), (8,11) sont des isthmes

FIG. 5.7.15 – Arbre couvrant en profondeur enraciné en *r* dans un graphe non-orienté connexe.

La valeur de $u(p)$ correspond à la valeur minimale entre les hauteurs des processeurs de $Arbre(p)$ et de leurs voisins hors-arbre. En fait, nous utiliserons $u(p)$ pour savoir si p à un descendant relié par une arête hors-arbre à l'un de ses ancestres (*upgoing edge*). L'exemple fourni dans la figure 5.7.15 illustre cette notion. Par exemple, dans la figure 5.7.15, le processeur 12 a deux descendants : 16 et 19, les voisins hors-arbre de 12, 16 et 19 sont 12 et 1. Donc, $u(12) = \min(\{h(12), h(16), h(19)\} \cup \{h(12), h(1)\}) = \min(\{4,5,6,1\}) = 1$.

Nous allons maintenant montrer, en utilisant les théorèmes 5.7.1, 5.7.2 et la remarque 5.7.1 que, d'une part, si $\forall p \in V \setminus \{r\}$, p peut calculer $Fils(p)$, $h(p)$ et $u(p)$ et r peut calculer $Fils(r)$, alors nous pouvons identifier tous les points d'articulation de G. D'autre part, si $\forall p \in V \setminus \{r\}$, p peut calculer $u(p)$ et $Pere(p)$ où $Pere(p)$ est le père de p dans $Arbre(r)$, alors nous pouvons identifier tous les isthmes de G.

Propriété 5.7.4 $\forall p \in V$, *p est un* point d'articulation *si et seulement si p vérifie une de ces deux conditions :*

1. $(p = \text{r}) \wedge (|Fils(p)| \geq 2)$.
2. $(p \neq \text{r}) \wedge (\exists q \in Fils(p) :: u(q) \geq h(p))$.

Preuve. Tout d'abord, d'après le théorème 5.7.1, nous pouvons trivialement déduire que la condition 1 est équivalente à la proposition "*r* est un point d'articulation".

Considérons maintenant la condition 2. Soit $p \in V \setminus \{r\}$. Par définition, $u(p)$ est égal à la plus petite hauteur parmi les hauteurs de p et ses descendants dans $Arbre(r)$ ainsi que les hauteurs de leurs voisins hors-arbre. Supposons ensuite que p est un point d'articulation. D'après le théorème 5.7.1, nous savons qu'il existe au moins un fils de p dans $Arbre(r)$, noté q, tel que aucun processeur parmi q et ses descendants dans $Arbre(r)$ n'est lié par une arête hors-arbre à un ancêtre de p. Donc, chaque voisin hors-arbre de q et ses descendants est un descendant de p. D'où, $u(q) \geq h(p)$. Ainsi, si $p \neq r$ est un point d'articulation, alors la condition 2 est vérifiée. En utilisant des arguments similaires, nous pouvons déduire la réciproque. □

Propriété 5.7.5 $\forall e \in E$, $e = \{p,q\}$ *est un isthme si et seulement si* $e \in E_T \wedge (((Pere(p) = q) \Rightarrow (u(p) = h(p))) \wedge ((Pere(q) = p) \Rightarrow (u(q) = h(q))))$.

Preuve. Tout d'abord, nous pouvons trivialement affirmer que l'ensemble des isthmes du réseau est un sous-ensemble de E_T. En effet, chaque isthme $\{p,q\}$ appartient à tout chemin allant de p à q (resp. de q à p) dans G. Donc, les isthmes appartiennent forcément à tout arbre couvrant (par définition, un graphe partiel de G *connexe*).

Supposons maintenant, par contradiction, que l'arête $\{p,q\}$ telle que $Pere(p) = q$ (le cas $Pere(q) = p$ est symétrique) est un isthme et $h(p) \neq u(p)$. Dans ce cas, $h(p) > u(p)$ car $u(p) = \min_{x \in Arbre(p)}(\{h(y) :: \{x,y\} \in E \setminus E_T\} \cup \{h(x)\})$. Or, $h(p) > u(p)$ signifie qu'il existe une arête hors-arbre entre un ancêtre de p et un descendant de p (d'après la définition de $u(p)$ et la propriété 5.7.3). Donc, le graphe partiel $G' = (V, E \setminus \{\{p,Pere(p)\}\})$ est connexe, contradiction.

Finalement, supposons, par contradiction, que l'arête $\{p,q\}$ telle que $Pere(p) = q$ (le cas $Pere(q) = p$ est symétrique) n'est pas un isthme et $h(p) = u(p)$. Dans ce cas, il existe au moins un cycle dans G incluant l'arête $\{p,Pere(p)\}$. Comme $Arbre(r)$ est un arbre couvrant en profondeur, ce cycle est composé d'arêtes de l'arbre et d'une arête hors-arbre. Cette arête hors-arbre relie un descendant de p à un ancêtre de p. Donc $u(p) < h(p)$ (d'après la définition de $u(p)$), contradiction. □

La figure 5.7.16 illustre les propriétés 5.7.4 et 5.7.5. Tout d'abord, *r* est la racine de l'arbre couvrant en profondeur et *r* a un unique fils dans cet arbre, donc, *r* n'est pas un point d'articulation d'après le

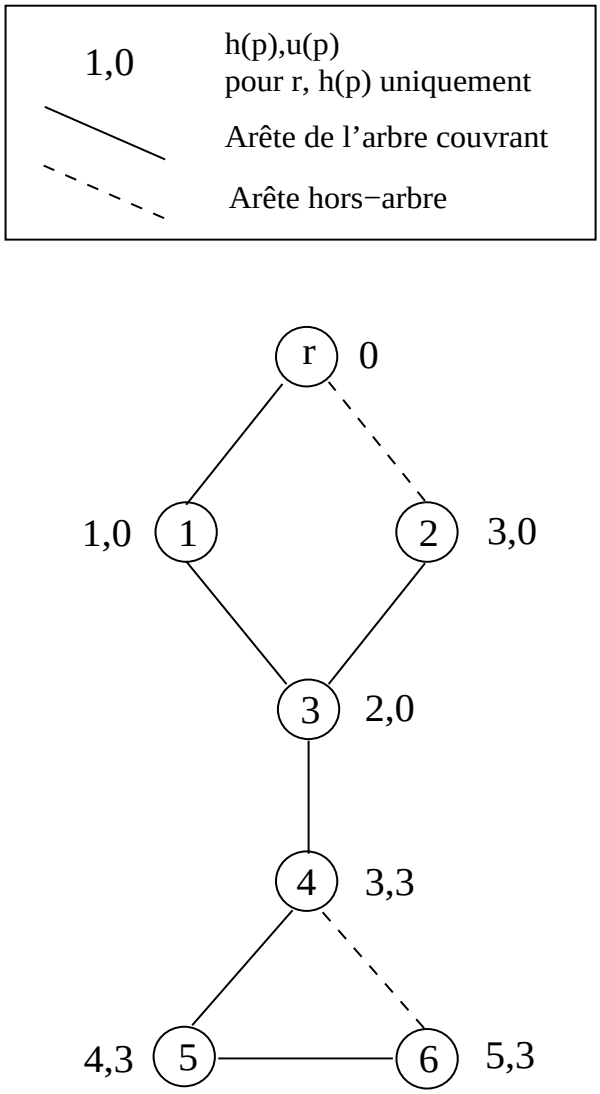


FIG. 5.7.16 – Exemple illustrant les propriétés 5.7.4 et 5.7.5.

point 1 de la propriété 5.7.4. Ensuite, le processeur 4 vérifie $u(4) \geq h(3)$ donc son père dans l'arbre, 3, est un point d'articulation d'après le point 2 de la propriété 5.7.4. Pour les mêmes raisons, 4 est un point d'articulation. Au contraire, le seul fils du processeur 1, le processeur 3, vérifie $u(3) < h(1)$ donc le processeur 1 n'est pas un point d'articulation. D'après le propriété 5.7.5, l'arête $\{3,4\}$ est un isthme car 4 vérifie $h(4) = u(4)$. En revanche, $h(5) \neq u(5)$, donc, l'arête $\{4,5\}$ n'est pas un isthme.

Nous allons maintenant présenter comment réaliser un protocole instantanément stabilisant d'identification des points d'articulation et des isthmes avec détection de terminaison à la racine en utilisant la composition conditionnelle entre un algorithme distribué calculant la fonction $u(p)$ (pour $p \neq r$), noté $\mathcal{UNNS}$ (*i.e.*, *Uppermost Non-Tree Neighbor of each Subtree*), et un protocole de parcours en profondeur instantanément stabilisant. Par souci de simplicité, nous allons présenter uniquement la composition, notée $\mathcal{UNNSDFS1}$, entre $\mathcal{UNNS}$ et $\mathcal{DFS1}$. Avec quelques modifications mineures, nous pouvons composer $\mathcal{UNNS}$ et $\mathcal{DFS2}$ pour obtenir le même résultat.

L'algorithme UNNSDFS1. Comme annoncé précédemment, $\mathcal{UNNSDFS1}$ (algorithme 5.7.9) est une composition conditionnelle entre $\mathcal{UNNS}$ et $\mathcal{DFS1}$. Le protocole $\mathcal{UNNS}$ (algorithmes 5.7.7

Algorithm 5.7.7 Algorithme $\mathcal{UNNS}$ pour $p = r$

Entrée :
$Neig_p$: ensemble des voisins (localement ordonné) ;
$Par_p \in Neig_p$: constante de $\mathcal{DFS}1$;
$Forward(p)$, $Backward(p)$: prédicats de $\mathcal{DFS}1$;
$Next_p$: macro de $\mathcal{DFS}1$;
Constante : $Level_p = 0$;
Variable : $CutNode_p$: booléen ;
Macros :
$Children_p \quad = \quad \{q \in Neig_p :: Par_q = p\}$;
$Update_p \quad = \quad$ **Si** $(Next_p = D)$ **alors** $CutNode_p := (|Children_p| \geq 2)$;
Règle :
$Forward(p) \vee Backward(p) \quad \rightarrow \quad Update_p$;

et 5.7.8) est un algorithme de calcul distribué des variables décrites dans les propriétés 5.7.4 et 5.7.5. Le fonctionnement du protocole $\mathcal{UNNSDFS}1$ (cf. algorithme 5.7.9) est présenté ci-dessous.

Tout d'abord, en utilisant la composition conditionnelle, les règles de $\mathcal{UNNS}$ sont exécutées dans les même mouvements que les règles F et B du protocole $\mathcal{DFS}1$ (cf. définition 5.7.1). La règle F de p est activable quand p vérifie $Forward(p)$. Respectivement, la règle B de p est activable quand p vérifie $Backward(p)$. Durant un parcours initié par la racine, quand un processeur $p \neq r$ reçoit le jeton pour la première fois en exécutant sa règle F, p peut effectuer dans $\mathcal{UNNS}$ un calcul dépendant de lui et de son père : un calcul *préfixe*. Ensuite, le parcours termine localement en p quand p exécute F ou B tandis que $Next_p = D$. p peut alors effectuer dans $\mathcal{UNNS}$ un calcul dépendant de ses descendants et/ou des ses voisins : un calcul *postfixe*.

Dans $\mathcal{UNNSDFS}1$, lorsque l'application le demande, la racine, r, initie en un temps fini un nouveau parcours. La requête correspond maintenant à une demande de marquage des points d'articulation et des isthmes du réseau.

À partir de l'action de démarrage (la règle F de r), chaque fois qu'un processeur $p \neq r$ reçoit le jeton courant pour la première fois (règle F), il évalue sa hauteur dans la variable $Level_p$ en fonction de la hauteur de P_p, *i.e.*, la hauteur de son père dans l'arbre couvrant. En fait, lorsque p exécute F, son père dans l'arbre couvrant est son unique prédécesseur (n.b., la règle F de p est activable seulement si p n'a qu'un prédécesseur). La variable S_p fournit en entrée des processeurs p tels que $p \neq r$ permet de connaître le prédécesseur de chaque processeur : le prédécesseur de p est son unique voisin q tel que $S_q = p$ lorsque p exécute F. Quand le parcours termine localement en q tel que $q \neq r$, q calcule $u(q)$ dans une variable $Back_q$ (n.b. la macro $Next_q$ fourni en entrée de $\mathcal{UNNS}$ permet à q de savoir lorsqu'il a localement terminé le parcours : quand q exécute les règles F ou B alors que $Next_q = D$). En effet, la valeur de $u(q)$ dépend des hauteurs de ses voisins q' qui sont maintenant stockées dans $Level_{q'}$ et des valeurs $u(q'')$ de ses descendants q'' dans le parcours qui ont déjà affecté leur variable $Back_{q''}$ à $u(q'')$. En particulier, si q est une feuille de l'arbre couvrant, q n'a pas de descendant et q évalue simplement sa variable $Back_q$ à partir des hauteurs de ses voisins. Connaissant maintenant $h(q)$ et $u(q)$, q peut alors décider (dans le même mouvement) :

- Il affecte à vrai sa variable $CutNode_q$ si et seulement s'il est un point d'articulation (conformément à la propriété 5.7.4).
- Il affecte à vrai sa variable $ParentLinkIsABridge_p$ si et seulement si l'arête $\{p,Par_p\}$ est un isthme (conformément à la propriété 5.7.5).

Enfin, lorsque le parcours termine en r, tous ses voisins ont été visités. r connaît donc l'ensemble de ses fils dans l'arbre couvrant en profondeur (la variable Par fourni entrée de tous les processeurs permet de calculer l'ensemble, noté $Children$, des fils d'un processeur dans l'arbre couvrant : l'ensemble des fils d'un processeur p est égal à l'ensemble de ses voisins q tels que $Par_q = p$) et décide en affectant vrai à la variable $CutNode_r$ si et seulement s'il est un point d'articulation (conformément

à la propriété 5.7.4).

Ainsi, quand $\mathcal{UNNSDFS}1$ termine à la racine, le système est dans une configuration où tous les points d'articulation et les isthmes du réseau sont marqués : pour chaque processeur p nous aurons $CutNode_p$ à vrai si et seulement si p est un point d'articulation et $ParentLinkIsABridge_p$ à vrai si et seulement si $\{p,Par_p\}$ est un isthme.

Algorithm 5.7.8 Algorithme $\mathcal{UNNS}$ pour $p \neq r$

Entrée :
$Neig_p$: ensemble des voisins (localement ordonné) ;
$S_p \in Neig_p \cup \{C,D\}$, $Par_p \in Neig_p$: variables de $\mathcal{DFS}1$;
$Forward(p)$, $Backward(p)$: prédicats de $\mathcal{DFS}1$;
$Next_p$: macro de $\mathcal{DFS}1$;
Variables : $Level_p$, $Back_p$: entiers ; $CutNode_p$, $ParentLinkIsABridge_p$: booléens ;

Macros :

P_p	=	$(q \in Neig_p :: S_q = p)$;
$Children_p$	=	$\{q \in Neig_p :: Par_q = p\}$;
$NonTreeLevel_p$	=	$\{Level_q :: q \in Neig_p \setminus (Children_p \cup \{Par_p \neq q\})\}$;
$ChildrenBack_p$	=	$\{Back_q :: q \in Children_p\}$
$ComputeBack_p$	=	$Back_p :=$ min$(ChildrenBack_p \cup NonTreeLevel_p \cup \{Level_p\})$;
$ComputeCutnode_p$	=	$CutNode_p := (\exists q \in Children_p :: Back_q \geq Level_p)$;
$ComputeBridge_p$	=	$ParentLinkIsABridge_p := (Back_p = Level_p)$;
$Update_p$	=	**Si** $(Next_p = D)$ **alors** $\{ComputeBack_p ; ComputeCutnode_p ; ComputeBridge_p\}$;

Règles :

$Forward(p)$	$\rightarrow$	$Level_p := Level_{P_p} + 1 ; Update_p$;
$Backward(p)$	$\rightarrow$	$Update_p$;

Algorithm 5.7.9 Algorithme $\mathcal{UNNSDFS}1$, $\forall p \in V$

$\mathcal{UNNS} \circ |_{\{Forward,Backward\}} \mathcal{DFS}1$

Preuve de la stabilisation instantanée. Nous allons maintenant prouver que $\mathcal{UNNSDFS}1$ est un protocole instantanément stabilisant avec détection de terminaison à la racine qui permet de marquer tous les points d'articulation et les isthmes du réseau. Pour cela, nous allons montrer que lorsqu'un parcours initié par la racine termine à la racine, $CutNode_p$ est vrai si et seulement si p est un point d'articulation et $ParentLinkIsABridge_q$ est vrai si et seulement si $\{q,Par_q\}$ est un isthme.

Nous rappelons que nous utiliserons les notations suivantes dans les preuves : lors d'un parcours initié par la racine, $\mathcal{DFS}1$ calcule un arbre couvrant en profondeur du réseau de hauteur H noté $Arbre(r) = (V,E_T)$ et $h(p)$ est la hauteur de p dans $Arbre(r)$, $\forall p \in V$.

Ce premier théorème montre que $\mathcal{UNNSDFS}1$ supporte le même démon que le protocole $\mathcal{DFS}1$, *i.e.*, le démon distribué inéquitable.

Théorème 5.7.3 *Sous un démon distribué inéquitable,* $\mathcal{UNNSDFS}1$ *est une composition* équitable *des protocoles* $\mathcal{UNNS}$ *et* $DFS1$.

Preuve. Les règles de $\mathcal{UNNS}$ sont activables seulement si la règle F ou la règle B de $DFS1$ est activable. Donc, par définition de la composition conditionnelle (définition 5.7.1, page 100) les règles de $\mathcal{UNNS}$ sont toujours exécutées dans les mêmes mouvements que les règles F et la règle B de $DFS1$. D'où, la composition est trivialement équitable. □

D'après le théorème 5.4.7 (page 64), nous savons que $\mathcal{DFS}1$ est un protocole instantanément stabilisant de parcours en profondeur. Plus précisément, à partir de n'importe quelle configuration initiale, à la suite d'une demande, la racine, *r*, initie en un temps fini un nouveau parcours. Durant ce parcours, tous les processeurs sont visités séquentiellement dans un ordre induit par la profondeur. Ensuite, d'après la propriété 5.7.2, nous savons que, $\forall p \in V$, à partir du moment où p est visité par le parcours initié, p ne participera plus jamais à un parcours corrompu. Enfin, comme $\mathcal{UNNS}$ n'écrit pas dans les variables de $\mathcal{DFS}1$ (par définition de la composition conditionnelle, page 100) et que la composition $\mathcal{UNNSDFS}1$ est équitable (d'après le théorème 5.7.3), nous savons que le protocole $\mathcal{UNNS}$ ne peut pas empêcher $\mathcal{DFS}1$ de fonctionner comme prévu (*i.e.*, conformément à ses spécifications). D'où, dans les preuves suivantes, nous allons uniquement observer le comportement du système à partir du moment où *r* initie le protocole $\mathcal{UNNSDFS}1$ et nous nous focaliserons uniquement sur le parcours réalisé à partir de *r* (nous ne considérerons pas les comportements anormaux relatifs au protocole $\mathcal{DFS}1$).

Dans le lemme suivant, nous affirmons que lorsqu'un parcours en profondeur initié par *r* termine, tout processeur p vérifie $Level_p = h(p)$ où $h(p)$ est la hauteur de p dans $Arbre(r)$, *i.e.*, l'arbre couvrant en profondeur calculé par ce parcours.

Lemme 5.7.1 *$\forall p \in V$, après que p ait reçu le jeton initié par* r *pour la première fois (règle F), p vérifie $Level_p = h(p)$ jusqu'à la fin du parcours.*

Preuve. Nous prouvons ce lemme par récurrence sur les hauteurs, $h(p)$, des processeurs p.

Soit $p \in V$ tel que $h(p) = 0$. Par définition, $p = r$. Or, $Level_r$ est une constante égale à 0. Donc, la récurrence est trivialement vérifiée pour $h(p) = 0$.

Supposons maintenant que, $\forall p \in V$ tel que $h(p) \leq k$ avec $0 \leq k < H$, après que p ait reçu le jeton initié par *r* pour la première fois, p vérifie $Level_p = h(p)$ jusqu'à la fin du parcours.

Soit $q \in V$ tel que $h(q) = k + 1$. Comme $q \neq r$, q reçoit par un voisin q' le jeton initié par la racine en un temps fini. Donc, q' est le père de q dans $Arbre(r)$ et $h(q') = k$. Par hypothèse de récurrence, $Level_{q'} = h(q')$ quand q reçoit le jeton en exécutant la règle F. D'où, quand q exécute F pour recevoir pour la première fois le jeton initié par la racine, il exécute aussi $Level_q := Level_{q'} + 1$ dans $\mathcal{UNNS}$ dans le même mouvement. Donc, q affecte $h(q)$ à $Level_q$. Ensuite, $Level_q$ n'est plus modifié jusqu'à la fin du parcours (cf. la propriété 5.7.2 et l'algorithme 5.7.8). D'où, la récurrence est vérifiée pour $h(q) = k + 1$. □

Dans le lemme suivant, nous affirmons que lorsqu'un parcours en profondeur initié termine, tout processeur p vérifie $Back_p = u(p)$. Pour montrer ce résultat, nous allons utiliser la notion de distance aux feuilles définie ci-dessous.

Définition 5.7.3 (Distance aux feuilles) *Soit p un processeur dans $Arbre(\mathrm{r})$. Soit f la feuille de $Arbre(p)$ de hauteur maximale. La distance aux feuilles de p, notée $d(p)$, est égale à $h(f) - h(x)$.*

Lemme 5.7.2 *$\forall p \in V \setminus \{\mathrm{r}\}$, après que le parcours initié par* r *soit terminé en p, p vérifie $Back_p = u(p)$ jusqu'à la fin du parcours.*

Preuve. Nous prouvons ce lemme par récurrence sur les distances aux feuilles, $d(p)$, $\forall p \in V \setminus \{r\}$.

Soit $f \in Arbre(r)$ tel que $d(f) = 0$. Par définition, f est une feuille de $Arbre(r)$. Quand f reçoit le jeton initié par *r* pour la première fois, f exécute sa règle F (dans $\mathcal{DFS}1$) et affecte D à S_f pour signifier que le parcours est localement terminé en f. Donc, d'après la propriété 5.7.1, tous ses voisins ont déjà été visités par le jeton et, d'après le lemme 5.7.1, f et ses voisins ont stocké leur hauteur respective dans leur variable $Level$. Dans le même mouvement que la règle F, f exécute la

macro $Update_f$. Or, comme $Next_f = D$, f exécute en particulier $ComputeBack_f$: $Back_f :=$ min ($ChildrenBack_f \cup NonTreeLevel_f \cup \{Level_f\}$). Or, $ChildrenBack_f = \emptyset$. Donc, $Back_f :=$ min ($NonTreeLevel_f \cup \{Level_f\}$) avec $NonTreeLevel_f = \bigcup_{y \in Neig_f \setminus \{Par_f\}} \{h(y)\}$ et $Arbre(f) = \{f\}$. Donc, $Back_f := \min_{x \in Arbre(f)} (\{ h(y) :: \{x,y\} \in E \setminus E_T\} \cup \{h(x)\}) = u(f)$. D'où, quand f exécute sa règle F, $u(f)$ est affecté à $Back_f$. Or, d'après l'algorithme 5.7.8 et par la propriété 5.7.2, nous savons que $Back_f$ ne sera plus modifié jusqu'à la fin du parcours. D'où, le lemme est vérifié pour tout f tel que $d(f) = 0$.

Supposons maintenant que, $\forall p \in V$ tel que $d(p) \leq k$ avec $0 \leq k < H - 1$, après que le parcours initié par *r* soit terminé en p, p vérifie $Back_p = u(p)$ jusqu'à la fin du parcours.

Soit $q \in V$ tel que $d(q) = k + 1$. Le parcours initié termine localement en q quand q affecte D à S_q en exécutant la règle B dans $\mathcal{DFS}1$ (cf. propriété 5.7.1). Donc, tous ses voisins ont déjà été visités par le jeton et, d'après le lemme 5.7.1, q et ses voisins ont stocké leur hauteur respective dans leur variable $Level$. Dans le même mouvement que la règle B, q exécute la macro $Update_q$. Or, comme $Next_q = D$, q exécute en particulier $ComputeBack_q$: $Back_q :=$ min ($ChildrenBack_q \cup NonTreeLevel_q \cup \{Level_q\}$). Par hypothèse de récurrence, $ChildrenBack_q = \bigcup_{z \in Children_q} \{u(z)\} = \bigcup_{z \in Fils(q)} \{\min_{x \in Arbre(z)}(\{h(y) :: \{x,y\} \in E \setminus E_T\} \cup \{h(x)\})\}$. Ensuite, $NonTreeLevel_q = (\bigcup_{y \in Neig_q \setminus Fils(q) \setminus \{Par_q\}} \{h(y)\}) = \{h(y) :: \{q,y\} \in E \setminus E_T\}$. Donc, $Back_q :=$ $\min(\bigcup_{z \in Fils(p)} \{\min_{x \in Arbre(z)}(\{h(y) :: \{x,y\} \in E \setminus E_T\} \cup \{h(x)\})\} \cup \{h(y) :: \{q,y\} \in E \setminus E_T\} \cup \{h(q)\}) = \min(\bigcup_{z \in Fils(q)} (\bigcup_{x \in Arbre(z)}(\{h(y) :: \{x,y\} \in E \setminus E_T\} \cup \{h(x)\})) \cup \{h(y) :: \{q,y\} \in E \setminus E_T\} \cup \{h(q)\})$. Or, $Arbre(q) = (\bigcup_{z \in Fils(q)} \{Arbre(z)\}) \cup \{q\}$. D'où, $Back_q := \min_{x \in Arbre(q)}(\{h(y) :: \{x,y\} \in E \setminus E_T\} \cup \{h(x)\}) = u(q)$. D'où, quand le parcours initié termine en q, $u(q)$ est affecté à $Back_q$. Or, d'après l'algorithme 5.7.8 et par la propriété 5.7.2, nous savons que $Back_q$ ne sera plus modifié jusqu'à la fin du parcours. L'égalité $Back_p = u(p)$ est donc est vérifiée pour $d(q) = k + 1$ et nous pouvons en déduire le lemme. □

Lemme 5.7.3 *À la terminaison d'un parcours initié, le système a atteint un point fixe où :*

- $\forall p \in V$, $CutNode_p = vrai$ *si et seulement si* p *est un point d'articulation.*
- $\forall e \in E$, $ParentLinkIsABridge_p = vrai$ *si et seulement si* $e = \{p,q\}$ *est un isthme et* $Pere(p) = q$.

Preuve. D'après les lemmes 5.7.1 et 5.7.2, lorsqu'un parcours initié termine localement en p avec $p \neq r$, nous avons $Level_p = h(p)$ et $Back_p = u(p)$. Or, d'après l'algorithme 5.7.8, les propriétés 5.7.4 et 5.7.5, les variables $CutNode_p$ et $ParentLinkIsABridge_p$ sont affectées correctement. Enfin, la propriété 5.7.2, nous assure que la valeur de ces variables ne sera plus changée.

De la même manière, lorsqu'un parcours initié termine à la racine, $Arbre(r)$ est entièrement calculé (n.b. la phase de nettoyage de $\mathcal{DFS}1$ ne réinitialise pas les pointeurs Par et la propriété 5.7.2 nous assure que la valeur de ces pointeurs n'a pas pu être modifiée par un quelconque comportement anormal). Donc, $Children_r$ correspond à l'ensemble des fils de *r* dans $Arbre(r)$. Or, d'après l'algorithme 5.7.7 et la propriété 5.7.4, *r* évalue $Cutpoint_r$ correctement. Enfin, la propriété 5.7.2, nous assure que la valeur de $Cutpoint_r$ ne sera plus changée.

D'où, dans la configuration où un parcours initié par *r* termine, nous avons :

- $\forall p \in V$, $CutNode_p = vrai$ si et seulement si p est un point d'articulation.
- $\forall e \in E$, $ParentLinkIsABridge_p = vrai$ si et seulement si $e = \{p,q\}$ est un isthme et $Pere(p) = q$.

□

D'après la remarque 3.2.1, le théorème 5.4.7 (page 64) et le lemme 5.7.3, nous déduisons le théorème suivant :

Théorème 5.7.4 *$\mathcal{UNSDFS}1$ est un protocole instantanément stabilisant avec détection de terminaison à la racine permettant l'identification des points d'articulation et des isthmes du réseau sous un démon distribué inéquitable.*

Nous allons maintenant nous intéresser aux ensembles séparateurs qui sont une généralisation des points d'articulation.

5.7.4 Ensemble de sommets séparateurs

Dans le domaine de la théorie des graphes, Provan et Ball ont prouvé que trouver les ensembles séparateurs minimaux d'un graphe quelconque est un problème NP-difficile [PB83]. Ainsi, quelques heuristiques pour des graphes quelconques [Kar00] et des méthodes complètes polynomiales pour des classes particulières de graphes [Ahm88, WSJ90] ont été développées. Ensuite, plusieurs travaux ont été réalisés dans le domaine des systèmes distribués non tolérants aux fautes [FL99, Rai82]. Ici, nous nous sommes intéressés au problème suivant : vérifier si un ensemble de processeurs donné est un séparateur du réseau. Ce problème est polynomial et, à notre connaissance, aucun protocole stabilisant n'a été écrit auparavant pour résoudre ce problème. Dans notre solution instantanément stabilisante l'ensemble à tester est fourni par l'application lors de la requête (par le bias le la macro $ApplicationRelease_r$, cf. la définition de la règle IR, sous-section 4.3.3, page 36). Une des propriétés les plus intéressantes de ce protocole est qu'en dépit de la configuration initiale du système, à la suite d'une demande, un cycle de calcul du protocole est initié en un temps fini par la racine, suite à cette initialisation, la racine décide en un temps fini si l'ensemble fourni en entrée du protocole est ou n'est pas un ensemble séparateur : cette décision est toujours juste en dépit de la configuration initiale. Donc, contrairement à une solution auto-stabilisante, une seule exécution du protocole est nécessaire pour que la racine décide correctement. Ce protocole a été présenté lors de la conférence internationale *HiPC'05* [CDV05a].

Notre protocole est basé sur l'approche suivante :

Approche. Soit $CS \subseteq V$. Soit $G' = (V', E')$ le sous-graphe de $G = (V, E)$ induit par $V' = V \setminus CS$. Soit $Arbre(r) = (V, E_T)$ un arbre couvrant en profondeur de G enraciné en r.

Notre protocole est basé sur la définition suivante (cette définition se déduit de la définition A.2.14, page 161).

Définition 5.7.4 *CS est un* ensemble séparateur *de G si et seulement si il existe au moins deux composantes connexes dans G'.*

Nous introduisons maintenant la notion de $CCRoot$, *i.e.*, une "racine" d'une composante connexe de G' (voir la figure 5.7.17).

Définition 5.7.5 (CCRoot) *Nous appellons* CCRoot *d'une composante connexe C de G', un sommet $p \in C$ vérifiant $h(p) \leq h(p')$, $\forall p' \in C$ (*i.e., *p est un sommet de C de hauteur minimale dans $Arbre(\mathrm{r})$).*

D'après la définition 5.7.5, nous pouvons trivialement déduire la remarque suivante.

Remarque 5.7.2 *Si* r $\notin CS$, *alors* r *est une $CCRoot$ de C où C est la composante connexe de G' contenant* r.

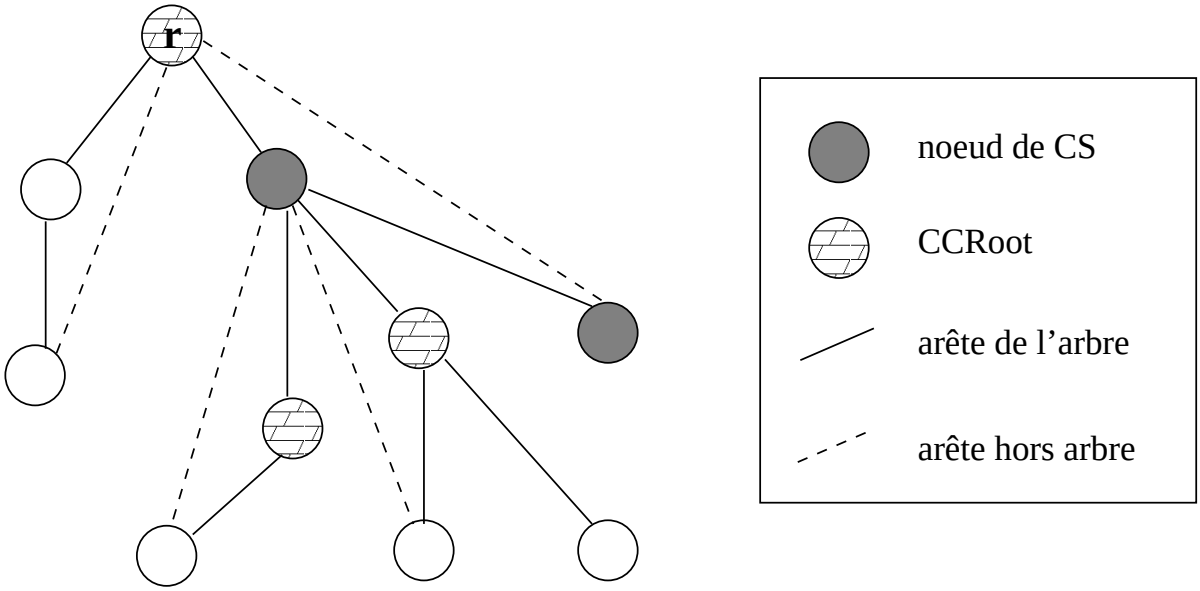


FIG. 5.7.17 – Exemple de $CCRoots$.

Les lemmes suivants montrent des propriétés utilisées par notre protocole.

Lemme 5.7.4 *Soit C une composante connexe de G' et p une $CCRoot$ de C. Le sous-arbre $Arbre(p)$ contient (au moins) tous les sommets de C.*

Preuve. Supposons, par contradiction, qu'il existe des sommets de C qui ne sont pas dans le sous-arbre $Arbre(p)$. Comme C est connexe, au moins un des sommets de $C \setminus Arbre(p)$, y, est un voisin d'un sommet x tel que $x \in Arbre(p)$. D'après la propriété 5.7.3 (page 101), soit $y \in Arbre(p)$ soit $p \in Arbre(y)$. Donc, par contradiction, $p \in Arbre(y)$, *i.e.*, $h(p) \geq h(y)$. Or, puisque p est une $CCRoot$ de C, d'après la définition 5.7.5, $h(y) \geq h(p)$. Donc, $h(y) = h(p)$ et $y = x$, contradiction. □

Corollaire 5.7.1 *Chaque composante connexe de G' contient une unique $CCRoot$.*

D'après la définition 5.7.4 et le corollaire 5.7.1, le théorème suivant est trivial.

Théorème 5.7.5 *CS est un* ensemble séparateur *de G si et seulement s'il existe au moins deux $CCRoots$ dans G'.*

Nous devons maintenant pouvoir détecter (localement) si un processeur est une $CCRoot$. Le lemme suivant nous donne un moyen de le faire.

Lemme 5.7.5 *Soit C une composante connexe de G'. Un sommet p est la $CCRoot$ de C si et seulement si p vérifie les deux conditions suivantes :*

- *$p \in C$.*
- *Pour tout processeur x de $Arbre(p) \cap C$, $\forall y \in Neig_x$: $(y \notin CS) \Rightarrow (h(y) \geq h(p))$.*

Preuve.

- **Si.** Supposons, par contradiction, que $p \in C$ et $\forall x \in Arbre(p) \cap C$, $\forall y \in Neig_x$: ($y \notin CS$) $\Rightarrow$ ($h(y) \geq h(p)$). Mais, p n'est pas une $CCRoot$. Donc, d'après la définition 5.7.5, il existe des processeurs q tels que $q \in C$ et $h(q) < h(p)$. En particulier, $q \notin Arbre(p)$. Mais, comme C est connexe, d'après la propriété 5.7.3, nous pouvons déduire que parmi ces processeurs q, au moins un, q', est un voisin du processeur x' tel que $x' \in Arbre(p) \cap C$. Or, par hypothèse, $h(q') \geq h(p)$, contradiction.
- **Seulement Si.** Supposons maintenant que p est une $CCRoot$ de C. Tout d'abord, d'après la définition 5.7.5, nous avons $p \in C$ et $\forall q \in C$, $h(q) \geq h(p)$. Or, $\forall x \in Arbre(p) \cap C$, $\forall y \in Neig_x$, si $y \notin CS$, alors $y \in C$ et $h(y) \geq h(p)$. Ainsi, $\forall x \in Arbre(p)$ tel que $x \in C$, $\forall y \in Neig_x : y \notin CS \Rightarrow h(y) \geq h(p)$.

□

Nous allons maintenant présenter comment réaliser un protocole instantanément stabilisant de détection d'ensembles séparateurs du réseau en utilisant la composition conditionnelle et les propriétés précédemment présentées. Le protocole est en fait la composition d'un protocole distribué, noté $\mathcal{CCRC}$ (*i.e., CCRoot Counting*), qui teste si un ensemble donné est un ensemble séparateur et un protocole de parcours en profondeur instantanément stabilisant. Ce protocole utilise les propriétés des arbres couvrants en profondeur afin de compter les $CCRoots$ du réseau. Par souci de simplicité, nous allons présenter uniquement la composition, noté $\mathcal{CCRCDFS1}$, entre $\mathcal{CCRC}$ et $\mathcal{DFS1}$. Avec quelques modifications mineures, nous pouvons composer $\mathcal{CCRC}$ et $\mathcal{DFS2}$ pour obtenir le même résultat.

L'algorithme CCRCDFS1. Comme annoncé précédemment, $\mathcal{CCRCDFS1}$ (algorithme 5.7.12) est une composition conditionnelle entre $\mathcal{CCRC}$ et $\mathcal{DFS1}$. Le protocole $\mathcal{CCRC}$ (algorithmes 5.7.10 et 5.7.11) est un algorithme distribué qui est basé sur le calcul pour chaque processeur p de la fonction $UNNTC_p$ (*i.e., Uppermost Non-Tree Neighbor of* $Tree(p) \cap C$) définie ci-dessous. Le calcul de cette variable lui permet d'appliquer le théorème 5.7.5 et le lemme 5.7.5 (cf. théorème 5.7.6).

Pour tout processeur p, la fonction $UNNTC(p)$ à la valeur suivante :
- **Si** $p \in CS$, **alors** $UNNTC(p) =$ -1.
- **Sinon**, p appartient à une composante connexe de G', noté C_p, et $UNNTC(p)$ est égale à la valeur minimale parmi les hauteurs de chaque sommet de $Arbre(p) \cap C_p$ et les hauteurs de leurs voisins hors-arbre q tels que $q \in C_p$.

Formellement, nous définissons $UNNTC(p)$ comme suit :

Définition 5.7.6 ($UNNTC(p)$**)** *Soit* $p \in V \setminus \{\mathrm{r}\}$. $UNNTC(p)$ *correspond à l'un des cas suivants :*
- $UNNTC(p) =$ *-1,* **si** *(*$p \in CS$*)*
- $UNNTC(p) = min_{x \in Arbre(p) \cap C_p}(\{h(x)\} \cup \{h(y) :: \{y, x\} \in E' \wedge h(y) < h(x)\})$ *où* C_p *est la composante connexe de* p *dans* G' *et* E' *l'ensemble des arêtes de* G'*,* **sinon**.

Le théorème suivant montre que si un processeur p connaît les valeurs de $h(p)$ et $UNNTC(p)$, alors il peut décider s'il est une $CCRoot$ ou pas.

Théorème 5.7.6 $\forall p \in V \setminus \{\mathrm{r}\}$, p *est une* $CCRoot$ *si et seulement si* $p \notin CS$ *et* $h(p) = UNNTC(p)$.

Preuve.
- **Si.** Supposons que $p \notin CS$ et $h(p) = UNNTC(p)$. D'après la définition 5.7.6, $h(p) = UNNTC(p)$ signifie que $\forall x \in Arbre(p) \cap C_p$, $\forall y \in Neig_x$ tel que $y \in C_p$, $h(y) \geq h(p)$ et, d'après le lemme 5.7.5, p est une $CCRoot$.

Algorithm 5.7.10 Algorithme $\mathcal{CCRC}$ pour $p = r$

Entrée :
$Neig_p$: ensemble des voisins (localement ordonné) ;
$S_p \in Neig_p \cup \{C,D\}$: variable de $\mathcal{DFS}1$;
$Forward(p)$, $Backward(p)$: prédicats de $\mathcal{DFS}1$;
$Next_p$: macro de $\mathcal{DFS}1$;
$InCS_p$: booléen ;
Constante : $Level_p = 0$;
Variables : $IsCutset_p$: booléen ; Cnt_p : entier ;
Macros :
$InitCnt_p$ = **si** ($InCS$) **alors** $Cnt_p := 0$; **sinon** $Cnt_p := 1$;
$UpdIsCutset_p$ = **si** ($Next_p = D$) **alors** $IsCutset_p := (Cnt_p \geq 2)$;
Règles :
$Forward(p) \rightarrow InitCnt_p ; UpdIsCutset_p$;
$Backward(p) \rightarrow Cnt_p := Cnt_{S_p} ; UpdIsCutset_p$;

Algorithm 5.7.11 Algorithme $\mathcal{CCRC}$ pour $p \neq r$

Entrée :
$Neig_p$: ensemble des voisins (localement ordonnée) ;
$S_p \in Neig_p \cup \{C,D\}$: variable de $\mathcal{DFS}1$;
$Forward(p)$, $Backward(p)$: prédicats de $\mathcal{DFS}1$;
$Next_p$: macro de $\mathcal{DFS}1$;
$InCS_p$: booléen ;
Variables : Cnt_p, $Level_p$, $Back_p$: entiers ;
Prédicat :
$IsCCRoot(p) \equiv (Back_p = Level_p)$
Macros :
P_p = $(q \in Neig_p :: S_q = p)$;
$NonCSAncLevel_p$ = $\{x \in \mathbb{N} :: (\exists q \in Neig_p :: (Level_q = x) \wedge (Level_q < Level_p) \wedge \neg InCS_q)\}$;
$NonCSDescBack_p$ = $\{x \in \mathbb{N} :: (\exists q \in Neig_p :: (Back_q = x) \wedge (Level_q > Level_p) \wedge \neg InCS_q)\}$;
$UpdBack_p$ = **si** ($InCS_p$) **alors** $Back_p := -1$;
sinon $Back_p := \min(\{Level_p\} \cup NonCSAncLevel_p \cup NonCSDescBack_p)$;
$UpdCnt_p$ = **si** ($IsCCRoot(p)$) **alors** $Cnt_p := Cnt_p + 1$;
$Update_p$ = **si** ($Next_p = D$) **alors** $\{UpdBack_p ; UpdCnt_p\}$;
Règles :
$Forward(p) \rightarrow Level_p := Level_{P_p} + 1 ; Cnt_p := Cnt_{P_p} ; Update_p$;
$Backward(p) \rightarrow Cnt_p := Cnt_{S_p} ; Update_p$;

- **Seulement Si.** Supposons, par contradiction, que p est une $CCRoot$ mais $p \in CS$ ou $h(p) \neq UNNTC(p)$. Puisque $p \notin CS$ (définition 5.7.5), nous avons, par contradiction, $h(p) \neq UNNTC(p)$. D'après la définition 5.7.6, $UNNTC(p) \leq h(p)$. Donc, par contradiction, $UNNTC(p) < h(p)$. $UNNTC(p) < h(p)$ signifie que $\exists\, x \in Arbre(p) \cap C_p$, $\exists\, \{y, x\} \in E'$ tel que $h(y) < h(p)$. Or, $x \in C_p$ et l'arête $\{x,y\}$ existe dans G'. Donc, $y \in C_p$ et, d'après la définition 5.7.5, $h(y) \geq h(p)$, contradiction.

□

Le protocole $\mathcal{CCRCDFS}1$ détecte si l'ensemble à tester, CS, est un ensemble séparateur en deux étapes :

– Il détecte les $CCRoots$ du réseau en calculant les valeurs de $h(p)$ et $UNNTC(p)$ de chaque processeur p.
– Puis, il les comptabilise et propage le résultat jusqu'à r pour qu'il décide en utilisant le théorème 5.7.5.

Nous décrivons maintenant le fonctionnement du protocole $\mathcal{CCRCDFS}1$.

Tout d'abord, comme pour le protocole précédent, dans $\mathcal{CCRCDFS}1$, les règles de $\mathcal{CCRC}$ sont

exécutées dans les même mouvements que les règles F et B du protocole $\mathcal{DFS}1$. Durant un parcours initié par la racine, un processeur p reçoit le jeton pour la première fois en exécutant sa règle F. Ensuite, le parcours termine localement en p quand p exécute F ou B tandis que $Next_p = D$. Pour simplifier le protocole, nous supposons que chaque processeur p sait s'il appartient à l'ensemble à tester (noté CS) grâce au booléen $InCS_p$. La variable $InCS_p$ est considérée comme une entrée de $\mathcal{CCRC}$. Cependant, il faut noter que nous devrions fournir CS uniquement en entrée de *r* (en utilisant un ensemble d'identités) lors de l'exécution de la règle externe IR (en utilisant la macro $ApplicationRelease_r$) et, ensuite, propager cet ensemble à tous les autres processeurs durant le parcours.

Lorsque l'application demande le test d'un ensemble CS, la racine, *r*, initie en un temps fini un nouveau parcours : la racine, *r*, commence un parcours en créant un jeton (règle F) et initialisant une variable Cnt_r. Cette variable est définie pour tous les processeurs et est utilisée pour compter les $CCRoots$ du réseau. Lorsque *r* initie un parcours, Cnt_r est initialisé à 0 ou 1 conformément à la remarque 5.7.2.

Ensuite, chaque fois qu'un processeur $p \neq r$ reçoit le jeton courant pour la première fois (règle F), il initialise Cnt_p en recopiant le compteur de son père ($Cnt_p := Cnt_{P_p}$) et calcule sa hauteur dans $Level_p$ (n.b. lorsque p exécute F, son père dans l'arbre couvrant est son unique prédécesseur P_p).

Chaque fois que le jeton revient en q (règle B), q met à jour Cnt_q en recopiant le compteur de son ancien successeur (S_p).

Quand le parcours termine localement en q tel que $q \neq r$, q calcule $UNNTC(q)$ dans $Back_q$ (n.b. la macro $Next_q$ fournie en entrée de $\mathcal{UNNS}$ permet à q de savoir lorsqu'il a localement terminé le parcours : quand q exécute les règles F ou B alors que $Next_q = D$). En effet, la valeur de $UNNTC(q)$ dépend des hauteurs de ses voisins q' qui sont maintenant stockées dans $Level_{q'}$ et des valeurs $u(q'')$ de ses descendants q'' dans le parcours qui ont déjà affecté leur variable $Back_{q''}$ à $u(q'')$. En particulier, si q est une feuille de l'arbre couvrant, q n'a pas de descendant et q évalue simplement sa variable $Back_q$ à partir des hauteurs de ses voisins. Connaissant maintenant $h(q)$ et $UNNTC(q)$, q peut alors décider (dans le même mouvement) s'il est ou pas une $CCRoot$: en appliquant le théorème 5.7.6, q incrémente Cnt_q si nécessaire.

Finalement, quand le parcours est complètement terminé (*i.e.*, le jeton est renvoyé à *r* et le jeton a visité tous ses processeurs), la variable Cnt_r contient le nombre de $CCRoots$ du réseau et, en appliquant le théorème 5.7.5, la racine affecte à vrai le booléen $IsCutset_r$ si et seulement si l'ensemble CS est un ensemble séparateur du réseau. Ainsi, lorsque *r* exécute l'action de terminaison de $\mathcal{CCRCDFS}1$ (règle T), nous savons si CS est un ensemble séparateur du réseau.

Algorithm 5.7.12 Algorithme $\mathcal{CCRCDFS}1$, $\forall p \in V$

$\mathcal{CCRC} \circ |_{\{Forward,Backward\}}\mathcal{DFS}1$

Ainsi, à partir d'une configuration initiale quelconque, à la terminaison d'un parcours initié par *r*, nous obtenons une configuration similaire à celle montrée dans la figure 5.7.18. Dans cet exemple, $CS = \{1, 6, 8\}$. Le système contient deux $CCRoots$: *r*, 2. La racine, *r*, est une $CCRoot$ car $r \notin CS$ (cf. remarque 5.7.2). Le processeur 2 est une $CCRoot$ car $2 \neq r$, $2 \notin CS$, et $Level_2 = Back_2$. Donc, durant le parcours, les variables Cnt comptent le nombre de $CCRoots$ du réseau (ici égales à 2) et $IsCutset_r$ reçoit la valeur vrai à la terminaison du parcours conformément au théorème 5.7.5.

Preuve de la stabilisation instantanée. Nous allons maintenant prouver que $\mathcal{CCRCDFS}1$ est un protocole instantanément stabilisant avec détection de terminaison à la racine qui permet de décider si un ensemble donné est un ensemble séparateur du réseau. Pour cela, nous allons montrer que

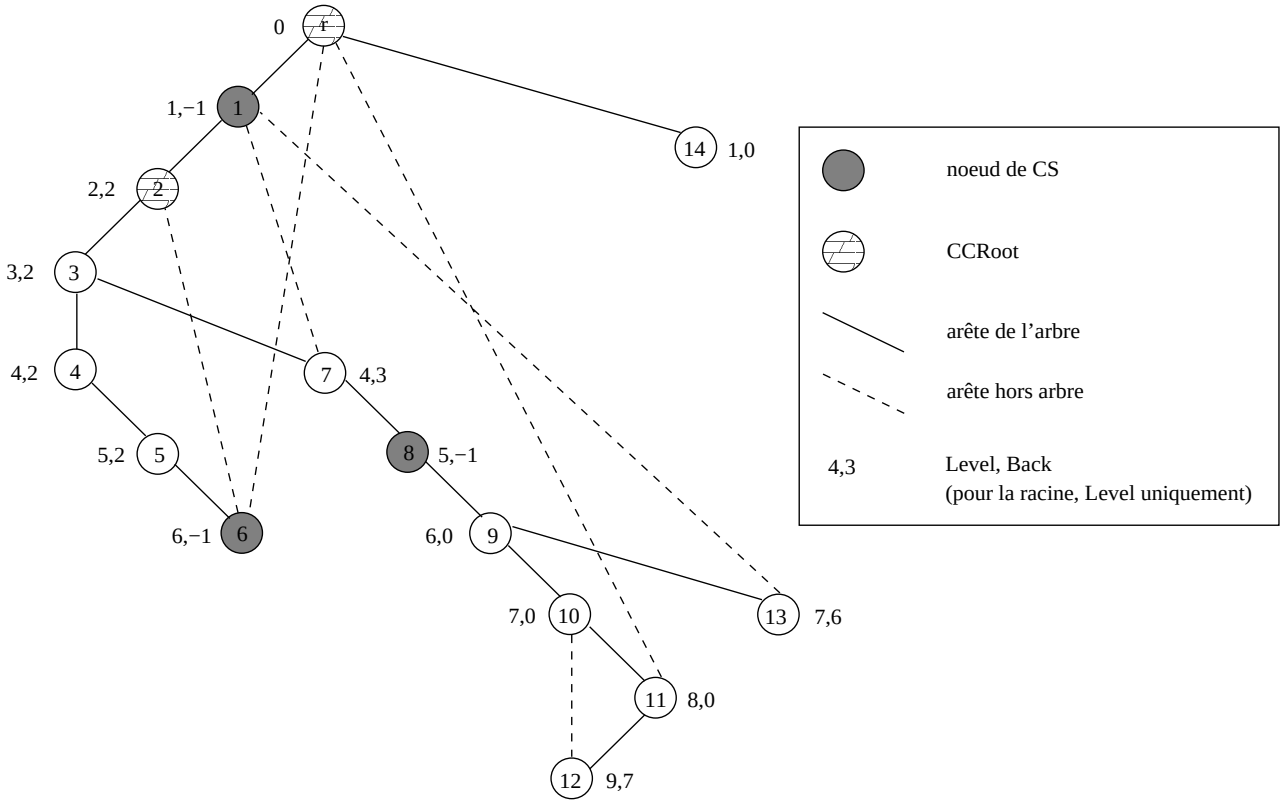


FIG. 5.7.18 – Exemple de configuration terminale de $\mathcal{CCRCDFS1}$.

lorsqu'un parcours initié termine à la racine, $IsCutset_r$ est vrai si et seulement si l'ensemble CS à tester est un ensemble séparateur.

Nous rappelons que nous utiliserons les notations suivantes dans les preuves :
- Lors d'un parcours initié par la racine, $\mathcal{DFS1}$ calcule un arbre couvrant en profondeur du réseau de hauteur H et noté $Arbre(r) = (V,E_T)$.
- $h(p)$ est la hauteur de p dans $Arbre(r)$, $\forall p \in V$.
- $CS \subseteq V$ est l'ensemble à tester (fourni par l'application).
- $G' = (V',E')$ est le sous-graphe de $G = (V,E)$ induit par $V' = V \setminus CS$.

Ce premier théorème montre que $\mathcal{CCRCDFS1}$ supporte le même démon que le protocole $\mathcal{DFS1}$, *i.e.*, le démon distribué inéquitable.

Théorème 5.7.7 *Sous un démon distribué inéquitable,* $\mathcal{CCRCDFS}1$ *est une composition* équitable *des protocoles* $\mathcal{CCRC}$ *et* $DFS1$.

Preuve. La preuve est identique à la preuve du théorème 5.7.3. □

En tenant compte du théorème précédent et suivant le même argumentaire que pour le protocole $\mathcal{UNNSDFS1}$, nous allons uniquement observer le comportement du système à partir du moment où *r* initie le protocole $\mathcal{CCRCDFS1}$ et nous nous focaliserons uniquement sur le parcours réalisé à partir de *r* pour prouver la stabilisation instantanée du protocole.

Dans le premier lemme, nous affirmons que quand un parcours en profondeur initié termine, tout processeur p vérifie $Level_p = h(p)$ où $h(p)$ est la hauteur de p dans $Arbre(r)$, *i.e.*, l'arbre couvrant en profondeur calculé par le parcours. Il se prouve par récurrence sur la hauteur des processeurs dans l'arbre couvrant de la même manière que le lemme 5.7.1, page 107.

Lemme 5.7.6 *$\forall p \in V$, après que p ait reçu le jeton initié par* r *pour la première fois, p vérifie $Level_p = h(p)$ et $Level_p$ reste égale à $h(p)$ jusqu'à la fin du parcours.*

Avec le lemme suivant, le lemme 5.7.6 et le théorème 5.7.6, $\forall p \in V \setminus \{r\}$, quand un parcours initié termine localement en p, p peut décider s'il est une $CCRoot$ ou pas. Le lemme 5.7.7 se prouve par récurrence sur la distance aux feuilles (définition 5.7.3, page 107) des processeurs dans l'arbre couvrant de la même manière que le lemme 5.7.2, page 107.

Lemme 5.7.7 *$\forall p \in V \setminus \{r\}$, un parcours initié termine localement en p, $Back_p = UNNTC(p)$.*

Le lemme suivant montre qu'à la terminaison de chaque parcours initié, la racine décide correctement.

Lemme 5.7.8 *À la fin d'un parcours initié, $IsCutset_r = true$ si et seulement si CS est un ensemble séparateur.*

Preuve. D'après le théorème 5.4.7 (page 64), nous savons que dans $\mathcal{DFS}1$, à la suite d'une demande à la racine, *r* initie un parcours en profondeur au cours duquel tous les processeurs sont séquentiellement visités dans un ordre induit par la profondeur d'abord. D'après la définition de la composition conditionnelle (définition 5.7.1, page 100), nous savons que les règles de $\mathcal{CCRC}$ ne peuvent pas empêcher le parcours de progresser correctement ($\mathcal{CCRC}$ n'écrit pas dans les variables de $\mathcal{DFS}1$). Donc, dans $\mathcal{CCRCDFS}$, suite à une demande à la racine, *r* crée un jeton en un temps fini. Dans le même mouvement, *r* exécute la macro $InitCnt_r$ dans $\mathcal{CCRC}$. Par $InitCnt_r$, *r* décide, conformément à la remarque 5.7.2, s'il est une $CCRoot$. Donc, *r* initialise Cnt_r correctement (cf. algorithme 5.4.3).

Ensuite, le jeton parcourt tout le réseau en profondeur d'abord. D'après les lemmes 5.7.6 et 5.7.7, quand le parcours termine localement en p tel que $p \in V \setminus \{r\}$, p vérifie à la fois $Level_p = h(p)$ et $Back_p = UNNC(p)$. Donc, conformément au théorème 5.7.6, le prédicat $IsCCRoot(p)$ détermine si p est une $CCRoot$ et p met à jour Cnt_p correctement par la macro $UpdBack_p$.

Enfin, chaque fois que le jeton se déplace, le nouveau porteur du jeton p met à jour Cnt_p avec le nombre de $CCRoots$ couramment détectées (cf. algorithmes 5.4.3 et 5.4.4).

D'où, quand le parcours termine en *r*, Cnt_r contient le nombre des $CCRoots$ du réseau et, d'après le théorème 5.7.5, *r* décide si CS est un ensemble séparateur et met à jour $IsCutset_r$ correctement (cf. $UpdIsCutset_r$ dans l'algorithme 5.4.3). □

D'après la remarque 3.2.1, le théorème 5.4.7 (page 64) et le lemme 5.7.8, nous déduisons le théorème suivant :

Théorème 5.7.8 *$\mathcal{CCRCDFS}1$ est instantanément stabilisant et détecte si CS est un ensemble séparateur en supposant un démon distribué inéquitable.*

5.7.5 Conclusion

Dans cette sous-section, nous avons présenté deux applications du parcours en profondeur utilisant des propriétés intrinsèques de ce type de parcours. Ces deux applications permettent de calculer des propriétés globales sur le réseau et utilisent un parcours en profondeur instantanément stabilisant. La première application (cf. section 5.7.3) est un calcul de point fixe avec détection de termination. L'algorithme présenté permet de marquer les *points d'articulation* et les *isthmes* du réseau. La seconde application (cf. section 5.7.4) permet d'évaluer si un ensemble fourni par une application est un *ensemble séparateur* du réseau. Les propriétés du protocole de parcours instantanément stabilisant permettent de calculer ces applications avec une détection de terminaison à la racine : à partir de

n'importe quelle configuration initiale, suite à une demande à la racine, le calcul demandé est initié en un temps fini et lorsque le calcul termine, la racine décide correctement. Contrairement à une solution auto-stabilisante, une seule exécution du protocole est nécessaire pour que la racine décide correctement. Il faut noter que les propriétés que nous utilisons dans ces protocoles peuvent être facilement adaptées pour des procotoles auto-stabilisants. Notamment, nous avons présenté un protocole silencieux auto-stabilisant d'identification des *points d'articulation* et des *isthmes* du réseau utilisant les mêmes propriétés que le protocole $\mathcal{UNNSDFS1}$ ([Dev05]). Grâce à la composition conditionnelle, les applications ont un surcoût en temps négligeable par rapport au protocole de parcours en profondeur utilisé. Ensuite, ces applications ne requièrent que $O(\log n)$ bits supplémentaires par processeur. Notamment, la complexité en espace de notre protocole d'identification des *points d'articulation* et des *isthmes* du réseau en fait le protocole le plus efficace. Enfin, nous pouvons remarquer que grâce aux propriétés de nos protocoles de parcours en profondeur instantanément stabilisant (en particulier, la propriété 5.7.1), ses applications sont aussi simples à réaliser que des solutions non-tolérantes aux fautes. En effet, les comportements anormaux dûs à la configuration initiale quelconque sont gérés par le protocole de parcours en profondeur instantanément stabilisant.

Chapitre 6

De l'algorithmique non tolérante aux fautes vers la stabilisation instantanée

Écrire et prouver des protocoles stabilisants (auto-stabilisants ou instantanément stabilisants) est généralement une tâche complexe. C'est pourquoi certains protocoles, appelés *transformateurs*, ont été proposés dans la littérature ([KP93, CDPV03]) afin de réaliser automatiquement une telle tâche. Le protocole de Katz et Perry ([KP93]) est basé sur un protocole de calcul d'état global du système (*snapshot*) et permet de transformer la plupart des protocoles non auto-stabilisants[1] en protocoles auto-stabilisants. Ce transformateur fonctionne dans le modèle à passage de messages. Dans [CDPV03], Cournier *et al* ont proposé des solutions instantanément stabilisantes pour quatre problèmes fondamentaux : *réinitialisation du réseau*, *calcul d'état global du système*, *élection de leader* et *détection de terminaison*. Ces solutions sont basées sur un protocole instantanément stabilisant de *propagation d'information avec retour* (PIR) pour réseaux quelconques (de tels protocoles sont proposés dans [BDPV99b, BDPV99c, CDV06b]). En utilisant ces protocoles fondamentaux, les auteurs montrent comment réaliser un protocole générique capable de fournir une version instantanément stabilisante de tout protocole pouvant être transformé par le protocole de Katz et Perry. Cependant, ce transformateur est écrit dans un modèle dont les hypothèses sont plus fortes que celui de Katz et Perry : le modèle à états.

Les transformateurs présentés dans [KP93, CDPV03] utilisent des mécanismes lourds pour transformer le protocole initial en un protocole stabilisant. De ce fait, le surcoût de la stabilisation est difficile à évaluer. En effet, ces transformateurs utilisent des calculs d'état global du système pour évaluer régulièrement un prédicat défini sur les variables du protocole à transformer. Ce prédicat caractérise les configurations normales du système. Cette technique permet de prévenir le système de tout interblocage ou famine. Les inconvénients majeurs de telles solutions sont les suivants :

1. Un tel prédicat est généralement difficile à formaliser.
2. Le nombre de fois où le transformateur lance un calcul d'état global du système peut ne pas être borné par rapport au temps d'exécution du protocole initial (*i.e.*, le protocole à transformer).

Nous proposons dans ce chapitre un protocole qui permet de transformer de façon semi-automatique des protocoles de service mono-initiateurs sans utiliser de calculs d'état global du système. Dans [CDV06a], nous montrons que ce transformateur permet de rendre instantanément stabilisant des protocoles à vagues auto-stabilisants simplement en ajoutant un booléen au niveau des actions de décision afin de pouvoir réintroduire la gestion explicite des requêtes au niveau du transformateur. Nous montrons ici qu'étant donné un protocole non résistant aux fautes il suffit de le modifier très légèrement de telle façon que l'initiateur puisse reprendre la main en un temps fini quelle que

[1] En fait, le transformateur de Katz et Perry peut rendre auto-stabilisant tout problème à suffixe fermé, cf. [KP93] pour plus de détails.

soit la configuration initiale du système. Cette modification repose sur une propriété que nous noterons *BreakingIn* et dont nous montrerons qu'elle est indispensable à la stabilisation. Autrement dit : écrire un protocole auto ou instantanément stabilisant est plus difficile qu'écrire un protocole vérifiant seulement *BreakingIn*. De plus, contrairement à [KP93, CDPV03], nous n'avons pas besoin d'utiliser dans le protocole un prédicat sur les configurations normales engendrées par le système et la technique utilisée par notre transformateur permet de calculer le surcoût raisonnable par rapport à l'algorithme initial. Enfin, l'algorithme transformé supporte les mêmes types de démons que l'algorithme initial contrairement à [CDPV03] pour lequel le transformé ne fonctionne au plus qu'avec un démon faiblement équitable même si l'algorithme initial fonctionne avec un démon inéquitable.

Un protocole de parcours en profondeur et un protocole de construction d'arbre en largeur illustreront la facilité avec laquelle nous pouvons rendre instantanément stabilisants ces protocoles grâce à notre transformateur. La différence notable entre les codes que nous obtenons dans ce chapitre et les solutions auto-stabilisantes ou instantanément stabilisantes de la littérature est suffisamment significative pour montrer que la propriété *BreakingIn* est non seulement plus facile d'un point de vue théorique mais qu'elle permet effectivement d'obtenir plus rapidement des codes très proches des algorithmes de base non tolérants aux fautes. Le dernier exemple que nous proposons, celui de l'exclusion mutuelle, nous permet de souligner une importante propriété de comptage due à notre transformateur. Cette propriété permet à tout processeur de distinguer un calcul juste d'un calcul qui pourrait être erroné. Nous obtenons alors de façon triviale un protocole d'exclusion mutuelle instantanément stabilisant en utilisant le protocole de circulation de jeton présenté dans ce même chapitre.

La suite du chapitre est organisée comme suit. Dans la section 6.1, nous présentons l'approche sur laquelle est basée notre transformateur. Ensuite, nous présentons l'outil qui est à la base de notre transformateur : la *propagation d'information avec retour* (section 6.2). Notre transformateur est décrit dans la section 6.3. Dans la section 6.4, nous présentons deux applications instantanément stabilisantes construites avec notre transformateur : un protocole de parcours en profondeur et un protocole de calcul d'arbre couvrant en largeur avec détection de terminaison à la racine. En utilisant le protocole de parcours en profondeur proposé dans la section 6.4, nous montrons dans la section 6.5 comment réaliser un protocole instantanément stabilisant d'exclusion mutuelle. Enfin, nous concluons dans la section 6.6.

6.1 Approche

Considérons un protocole de service mono-initiateur, $\mathcal{A}$, (n.b. r est l'unique initiateur) résolvant une tâche spécifique $\mathcal{TASK}$ dans un environnement sans faute. De plus, nous supposons que les règles de décision n'existent que dans le code de l'initiateur (par exemple, tout protocole de circulation de jeton vérifie ces contraintes). Nous voulons transformer $\mathcal{A}$ en protocole instantanément stabilisant sans utiliser de calculs d'état global du système. Dans [KP93, CDPV03], les calculs d'état global du système sont utilisés pour prévenir le système de tout interblocage ou famine. Puisque nous n'utilisons pas de calculs d'état global du système, nous ne pouvons pas détecter les interblocages et les famines qui peuvent apparaître quand $\mathcal{A}$ est exécuté à partir d'une configuration initiale quelconque. Nous proposons donc de transformer (légèrement) $\mathcal{A}$ en un protocole $\mathcal{B}$ vérifiant une propriété supplémentaire. Cette propriété doit être suffisante pour permettre à $\mathcal{B}$ d'être transformé automatiquement en un protocole instantanément stabilisant sans utiliser de calculs d'état global du système. Cette propriété doit être la plus simple possible. En particulier, elle doit être plus simple à obtenir que l'auto-stabilisation ou la stabilisation instantanée. Dans la suite, nous noterons $\mathcal{SSBB}(\mathcal{B})$ (*i.e.*, *Snap-Stabilizing Black Box*) la version instantanément stabilisante de $\mathcal{B}$ obtenue avec notre transformateur. D'après la remarque 3.2.1 (page 25), le code de $\mathcal{B}$ doit assurer la propriété suivante :

– À partir d'une configuration quelconque et suite à une demande, r initie en un temps fini

$\mathcal{SSBB}(\mathcal{B})$. (*Service I*)
- Dès que $\mathcal{SSBB}(\mathcal{B})$ est initié, $\mathcal{TASK}$ est exécutée conformément à ses spécifications. (*Service II*)

Tout d'abord, d'après (*Service I*), à partir d'une configuration quelconque, le système doit atteindre une configuration à partir de laquelle $\mathcal{SSBB}(\mathcal{B})$ peut démarrer correctement. Cela implique que quand la racine reçoit une demande d'exécution de $\mathcal{SSBB}(\mathcal{B})$ (*i.e.*, de $\mathcal{A}$), $\mathcal{SSBB}(\mathcal{B})$ doit démarrer en un temps fini mais sans abandonner un calcul de la tâche $\mathcal{TASK}$ précédemment initié. Une façon d'obtenir une telle propriété est de concevoir dans $\mathcal{B}$ un prédicat pour *r*, noté $\mathcal{B}.End(r)$, qui vérifie la propriété *BreakingIn* définie ci-dessous (cette propriété utilise une notion de *stabilité* définie en préambule) :

Définition 6.1.1 (Stable) *Soient $\mathcal{P}$ un protocole, $\mathcal{X}$ un prédicat défini sur les variables de $\mathcal{P}$ et $\mathcal{D}$ un démon. $\mathcal{X}$ vérifie* Stable*($\mathcal{D}$) ($\mathcal{X}$ est dit* stable *pour le démon $\mathcal{D}$) si et seulement si : $\mathcal{X}$ est vérifié implique que s'il existe des règles de $\mathcal{P}$ ayant $\mathcal{X}$ comme garde, alors au moins une de ces règles finit par être exécutée en dépit du démon $\mathcal{D}$.*

Définition 6.1.2 (BreakingIn) *Soit $\mathcal{P}$ un protocole de service mono-initiateur (*r *est l'unique initiateur) et $\mathcal{X}$(r) un prédicat défini sur les variables de* r *et ses voisins dans $\mathcal{P}$, le prédicat $\mathcal{X}$(r) vérifie* BreakingIn*($\mathcal{D}$) si et seulement si :*

1. *À partir de n'importe quelle configuration, $\mathcal{X}$(r) est vérifié en un temps fini en dépit du démon $\mathcal{D}$.*
2. *Si $\mathcal{X}$(r) est vérifié et qu'un cycle de calcul a été précédemment initié, alors ce cycle de calcul est terminé (*i.e.*,* r *a décidé dans ce cycle).*
3. *Dans toute configuration où $\mathcal{X}$(r) est vérifié, aucune règle de* r *dans $\mathcal{P}$ n'est activable.*
4. *$\mathcal{X}$(r) vérifie* Stable*($\mathcal{D}$).*

Remarque 6.1.1 *Par souci de concision, dans la suite nous utiliserons les abréviations suivantes :*
- *DSF (i.e.,* distributed strongly fair*) pour le démon distribué fortement équitable.*
- *DWF (i.e.,* distributed weakly fair*) pour le démon distribué faiblement équitable.*
- *DUF (i.e.,* distributed unfair*) pour le démon distribué inéquitable.*

Le résultat suivant est la conséquence des points 1 et 4 de la définition 6.1.2.

Conséquence 6.1.1 *Soit $\mathcal{P}$ un protocole de service mono-initiateur (*r *est l'unique initiateur) et $\mathcal{X}$(r) un prédicat vérifiant* BreakingIn*($\mathcal{D}$) défini sur les variables de* r *et ses voisins dans $\mathcal{P}$. Soit $e = \gamma_0$, γ_1, ..., γ_k, ... une exécution de $\mathcal{P}$ sous le démon $\mathcal{D}$.*
- *Si $\mathcal{D} = DSF$, alors pour tout suffixe non vide e' de e, $\exists \gamma_i \in e'$ telle que $\mathcal{X}$(r) est vérifié dans γ_i.*
- *Si $\mathcal{D} = DWF$, alors $\exists \gamma_i \in e$ telle que $\forall j \geq i$, $\mathcal{X}$(r) est vérifié dans γ_j.*
- *Si $\mathcal{D} = DUF$, alors e est fini et $\mathcal{X}$(r) est vérifié dans sa configuration terminale.*

Dans le paragraphe suivant, nous allons montrer que la propriété *BreakingIn* est nécessaire dans les protocoles de service stabilisants (auto-stabilisants ou instantanément stabilisants) mono-initiateurs. Ainsi, nous pourrons conclure qu'il est plus facile d'écrire un protocole vérifiant *BreakingIn* qu'un protocole stabilisant.

BreakingIn et protocoles stabilisants. Soit $\mathcal{P}$ un protocole de service stabilisant mono-initiateur dont toutes les règles de décision sont uniquement exécutées par l'initiateur (n.b. $\mathcal{P}$ est auto-stabilisant ou instantanément stabilisant). Dans notre modèle de calcul avec gestion explicite des requêtes, chaque cycle de calcul de $\mathcal{P}$ (conforme ou non aux spécifications) termine en un temps fini par l'exécution d'une règle de décision de l'initiateur.

Si $\mathcal{P}$ est instantanément stabilisant, alors suite à une décision, le système atteint en un temps fini une configuration γ où plus aucune règle de l'initiateur dans $\mathcal{P}$ n'est activable jusqu'à l'exécution de la règle externe IR : l'initiateur atteint un état local où il est prêt à démarrer un nouveau cycle de calcul et où il est en attente de la prochaine requête. Soit $\mathcal{X}(r)$ un prédicat caractérisant cet état local de l'initiateur. D'après la définition 6.1.2, $\mathcal{X}(r)$ vérifie *BreakingIn*$(\mathcal{D})$ où $\mathcal{D}$ est le démon supporté par $\mathcal{P}$.

Si $\mathcal{P}$ est auto-stabilisant, $\mathcal{P}$ a besoin d'être répété perpétuellement. Cela implique que le prédicat $ApplicationRequest(r)$ de la garde de sa règle IR (*i.e.*, la règle externe gérant la requête) est toujours vrai : $ApplicationRequest(r) \equiv true$. Cependant, après la décision du cycle de calcul précédent, l'initiateur ne peut redémarrer un nouveau cycle que si la règle IR a été exécutée. Donc, le système atteint en un temps fini une configuration γ où l'activation des actions de démarrage est conditionnée à l'exécution préalable de la règle IR, *i.e.*, sans l'intervention extérieure de la règle IR, plus aucune règle de l'initiateur dans $\mathcal{P}$ n'est activable. Soit $\mathcal{X}(r)$ un prédicat caractérisant l'état local de l'initiateur dans γ. Comme pour un protocole instantanément stabilisant, $\mathcal{X}(r)$ caractérise l'état local vérifié par l'initiateur lorsqu'il est prêt à démarrer un nouveau cycle de calcul et qu'il attend la prochaine requête. D'après la définition 6.1.2, $\mathcal{X}(r)$ vérifie *BreakingIn*$(\mathcal{D})$ où $\mathcal{D}$ est le démon supporté par $\mathcal{P}$.

D'après la discussion ci-dessus, nous déduisons le théorème suivant :

Lemme 6.1.1 *Pour tout protocole de service mono-initiateur stabilisant $\mathcal{P}$ dont les règles de décision sont exécutées par l'initiateur uniquement, il existe un prédicat défini sur les variables de l'initiateur et de ses voisins qui vérifie* BreakingIn$(\mathcal{D})$.

D'après le lemme 6.1.1, nous pouvons affirmer que la propriété *BreakingIn* est indispensable dans les protocoles de service stabilisants mono-initiateurs. Dans le lemme suivant, nous affirmons que la réciproque du lemme 6.1.1 est fausse.

Lemme 6.1.2 *Il existe des protocoles de service mono-initiateurs non stabilisants $\mathcal{P}$ dont les règles de décision sont exécutées par l'initiateur uniquement pour lesquelles il existe un prédicat défini sur les variables de l'initiateur et de ses voisins qui vérifie* BreakingIn$(\mathcal{D})$.

Preuve. Deux protocoles non stabilisant vérifiant *BreakingIn*$(\mathcal{D})$ sont présentés dans la section 6.4. □

D'après les lemmes 6.1.1 et 6.1.2, nous pouvons déduire qu'il est plus facile d'écrire des protocoles vérifiant *BreakingIn* que des protocoles stabilisants :

Théorème 6.1.1 *La propriété* BreakingIn *est strictement plus faible que la stabilisation.*

Propriété SSBB-Friendly. Si nous supposons maintenant l'existence du prédicat $\mathcal{B}.End(r)$ (*i.e.*, un prédicat de *r* vérifiant $BreakingIn(\mathcal{D})$), nous avons juste besoin de réinitialiser les variables de $\mathcal{B}$ dès que $\mathcal{B}.End(r)$ est satisfait afin de vérifier *Service II* (n.b. *BreakingIn*$(\mathcal{D})$ assure que $\mathcal{B}.End(r)$ finit par être vérifié en dépit du démon). À cet effet, nous supposons que, pour tout processeur p, toutes les affectations de variables nécessaires pour générer une *configuration initiale normale* de $\mathcal{A}$ sont stockées dans une macro de $\mathcal{B}$ notée $\mathcal{B}.Init_p$. Par souci de clarté, nous notons $\mathcal{B}.Init$ l'ensemble

des macros $\mathcal{B}.Init_p$. En utilisant $\mathcal{B}.Init$, une réinitialisation du système est réalisée au démarrage de $\mathcal{SSBB}(\mathcal{B})$ et, dès que cette réinitialisation est terminée, $\mathcal{B}$ exécute la tâche $\mathcal{TASK}$ comme $\mathcal{A}$ dans un environnement sans faute. En particulier, cela signifie que les actions de démarrage de $\mathcal{SSBB}(\mathcal{B})$ correspondent aux actions de démarrage de la réinitialisation et que les actions de démarrage de la réinitialisation doivent prendre en compte les requêtes pour $\mathcal{B}$ (en utilisant $\mathcal{B}.Request_r$) à la place de $\mathcal{B}$ lui-même. Ainsi, nous pouvons maintenant définir la propriété *SSBB-Friendly* qui est la propriété que $\mathcal{B}$ doit vérifier pour être transformé par $\mathcal{SSBB}$.

Définition 6.1.3 (SSBB-Friendly) *Soit $\mathcal{P}$ un protocole de service mono-initiateur (*r *est l'unique initiateur). $\mathcal{P}$ vérifie* SSBB-Friendly($\mathcal{TASK}$,$\mathcal{D}$) *où $\mathcal{TASK}$ est une tâche spécifique et $\mathcal{D}$ un démon si et seulement si :*

1. *$\mathcal{P}$ contient un prédicat $\mathcal{P}.End$(r) vérifiant $BreakingIn(\mathcal{D})$.*
2. *$\mathcal{P}$ contient un ensemble de macros $\mathcal{P}.Init$ tel que, à partir de toute configuration engendrée par $\mathcal{P}.Init$, $\mathcal{P}$ exécute $\mathcal{TASK}$.*
3. *Aucune règle de* r *dans $\mathcal{P}$ ne réfère à $\mathcal{P}.Request_r$.*

À la suite d'une requête, $\mathcal{SSBB}(\mathcal{B})$ réinitialise (en utilisant $\mathcal{B}.Init$) en un temps fini (n.b. la réinitialisation démarre en un temps fini grâce à $\mathcal{B}.End(r)$) les variables de $\mathcal{B}$ afin que le système atteigne une *configuration initiale normale* de $\mathcal{A}$ puis, $\mathcal{SSBB}(\mathcal{B})$ laisse le contrôle de l'exécution à $\mathcal{B}$ pour qu'il réalise la tâche $\mathcal{TASK}$ comme $\mathcal{A}$ dans un environnement sans faute. Une technique bien connue pour réaliser une réinitialisation globale dans un système distribué est basée sur la *propagation d'information avec retour* (PIR). Plusieurs protocoles de PIR pour réseaux quelconques ont déjà été proposés dans le domaine de l'auto-stabilisation ([AKY90, APSV91, AKM$^+$93, Var93, AG94]) ainsi que dans celui de la stabilisation instantanée ([CDPV02, BCV03, CDV06b]). Tout protocole de PIR suit le schéma suivant : l'initiateur démarre le protocole par la diffusion d'un message *m* (*phase de diffusion*), ensuite, chaque processeur non-initiateur envoie en un temps fini un récepissé à l'initiateur attestant de la réception de *m* (*phase de retour*). Un exemple d'exécution de PIR non-tolérant aux fautes est présenté dans la figure 6.1.1.

En utilisant un protocole de PIR, une réinitialisation du réseau peut être réalisée comme suit :

1. Tout d'abord, l'initiateur diffuse un message d'abandon dans le réseau.
2. À la réception de ce message, les processeurs stoppent l'exécution locale de $\mathcal{B}$.
3. Finalement, les processeurs réinitialisent les variables de $\mathcal{B}$ durant la phase de retour.

Comme nous souhaitons appliquer notre technique quel que soit le type de démon, nous avons besoin d'un protocole de PIR fonctionnant avec le démon le plus général du modèle à états : le démon distribué inéquitable. Un tel protocole est fourni dans [CDV06b] et sera noté $\mathcal{PIR}$. Dans la section suivante, nous présentons le principe de fonctionnement du protocole $\mathcal{PIR}$ (*i.e.*, le protocole fourni dans [CDV06b]), puis, nous rappelons certaines de ses propriétés (ses propriétés seront utilisées lors de la preuve de stabilisation instantanée de notre transformateur).

6.2 L'outil de base : le PIR

Le protocole $\mathcal{PIR}$ vérifie la spécification suivante :

1. À partir de n'importe quelle configuration, quand *r* a un message à diffuser, *r* initie la diffusion de *m* en un temps fini.
2. Suite à cette initialisation, tous les autres processeurs recoivent *m* puis envoient un récepissé attestant de la réception de *m*, ce récepissé atteint ensuite *r* en un temps fini.

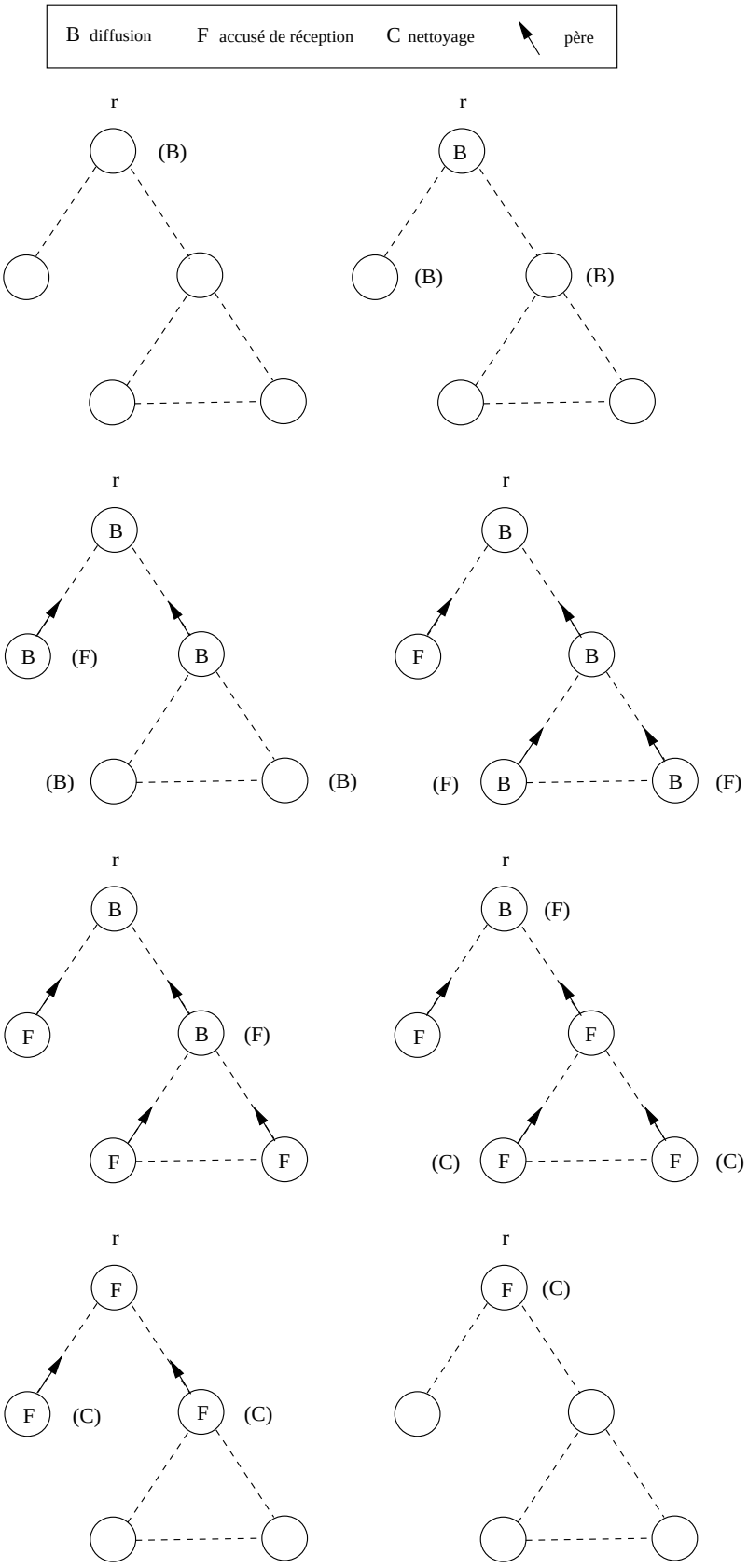


FIG. 6.1.1 – Un exemple de PIR non-tolérant aux fautes.

Algorithm 6.2.13 Algorithme $\mathcal{PIR}$ pour $p = r$

Entrée-sortie : $Request_p \in \{Wait,In,Out\}$;

Entrée : $Neig_p$: ensemble des voisins (localement ordonné) ;

Constantes : $Par_p = \perp$; $L_p = 0$;

Variables : $PIF_p \in \{B,F,PC,C\}$; $Que_p \in \{Q,R,A\}$;

Macro :

$Child_p = \{q \in Neig_p :: (PIF_q \neq C) \wedge (Par_q = p) \wedge (L_q = L_p + 1)$
$\wedge [(PIF_q \neq PIF_p) \Rightarrow (PIF_p \in \{B,PC\} \wedge PIF_q = F)]\}$;

Prédicates généraux :

$CFree(p) \equiv (\forall q \in Neig_p :: PIF_q \neq C)$
$Leaf(p) \equiv [\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Par_q \neq p)]$
$BLeaf(p) \equiv (PIF_p = B) \wedge [\forall q \in Neig_p :: (Par_q = p) \Rightarrow (PIF_q = F)]$
$AnswerOk(p) \equiv (Que_p = A) \wedge [\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q = A)]$

Gardes :

$Broadcast(p) \equiv (PIF_p = C) \wedge Leaf(p)$
$Feedback(p) \equiv BLeaf(p) \wedge CFree(p) \wedge AnswerOk(p)$
$PreClean(p) \equiv (PIF_p = F) \wedge [\forall q \in Neig_p :: (Par_q = p) \Rightarrow (PIF_q \in \{F,C\})]$
$Cleaning(p) \equiv (PIF_p = PC) \wedge Leaf(p)$
$Require(p) \equiv (PIF_p \in \{B,F\}) \wedge [(PIF_p = B) \Rightarrow CFree(p)]$
$\wedge [[(Que_p = Q) \wedge (\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q \in \{Q,R\}))]$
$\vee [(Que_p = A) \wedge (\exists q \in Neig_p :: (PIF_q \neq C) \wedge ((Que_q = Q)$
$\vee (q \in Child_p \wedge Que_q = R)))]]$
$Answer(p) \equiv (PIF_p \in \{B,F\}) \wedge [(PIF_p = B) \Rightarrow CFree(p)] \wedge (Que_p = R)$
$\wedge (\forall q \in Child_p :: Que_q \in \{W,A\})$
$\wedge [\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q \neq Q)]$

Règles :

Phase de PIR :

Bst	::	$Broadcast(p) \wedge (Request_p = Wait)$	$\rightarrow$	$PIF_p := B ; Que_p := Q ;$ $Request_p := In ;$	/* Démarrage */
Fck	::	$Feedback(p)$	$\rightarrow$	$PIF_p := F ;$	
PC	::	$PreClean(p)$	$\rightarrow$	$PIF_p := PC ;$	
C	::	$Cleaning(p)$	$\rightarrow$	$PIF_p := C ;$	
T	::	$Broadcast(p) \wedge (Request_p = In)$	$\rightarrow$	$Request_p := Out ;$	/* Terminaison */

Phase de Question :

QR	::	$Require(p)$	$\rightarrow$	$Que_p := R ;$
QA	::	$Answer(p)$	$\rightarrow$	$Que_p := A ;$

6.2.1 Protocole

Le protocole $\mathcal{PIR}$ (cf. algorithmes 6.2.13 et 6.2.14) utilise les mêmes mécanismes que le protocole $\mathcal{DFS}2$ (section 5.5, page 69). Donc, la preuve de la stabilisation instantanée du protocole $\mathcal{PIR}$ est similaire à celle de $\mathcal{DFS}2$ (sous-section 5.5.2) et n'est pas présentée dans cette thèse (la preuve complète de ce protocole est disponible dans le rapport interne [CDV06c]). En fait, dans le protocole $\mathcal{PIR}$, nous avons juste remplacé la phase de visite de $\mathcal{DFS}2$ par une phase de PIR, elle-même décomposée en deux sous-phases : une *phase de diffusion* et une *phase de retour*. Les phases de diffusion et de retour sont exécutées successivement mais les processeurs exécutent ces phases en parallèle comme montré dans la figure 6.1.1 et ce contrairement à la phase de visite du protocole $\mathcal{DFS}2$ qui est exécutée de manière séquentielle.

Le protocole $\mathcal{PIR}$ utilise 4 variables :

– Les variables Par, Que et L ont un rôle identique à celles du protocole $\mathcal{DFS}2$: Par est un pointeur père qui est affecté lors de la phase de diffusion (le père d'un processeur $p \neq r$ est le processeur par qui p a reçu le message), la variable Que gère les questions et la variable de niveau L permet de connaître la hauteur d'un processeur dans l'arbre couvrant construit par la phase de diffusion afin de casser les cycles possibles dûs à la variable Par dans la configuration

initiale.

– La variable PIF, quant à elle, remplace le pointeur de successeur S mais a un rôle similaire : la variable PIF permet, comme S, de faire progresser un calcul issu de la racine mais de manière parallèle (alors que S fait progresser le calcul de manière séquentielle). PIF est défini pour chaque processeur comme suit : $PIF_r \in \{B,F,PC,C\}$ et $PIF_p \in \{B,F,PC,C,EB,EF\}$ pour $p \neq r$. $PIF_p = B$ (*Broadcast*) signifie que le processeur p participe à une phase de diffusion. $PIF_p = F$ (*Feedback*) quand p est dans une phase de retour. $S_p = C$ (*Clean*) signifie que p ne participe à aucun cycle de PIR. L'état PC (*PreClean*) est utilisé lors de la phase de nettoyage comme dans le protocole $\mathcal{DFS}2$. Enfin les états EB (*ErrorBroadcast*) et EF (*ErrorFeedback*) sont utilisés pour le gel des processus anormaux lors de la phase de correction.

En utilisant les variables précédemment décrites, le protocole $\mathcal{PIR}$ fonctionne comme suit. À la suite d'une demande (exécution de la règle IR, *i.e.*, $Request_r := Wait$), la racine exécute sa règle Bst (l'action de démarrage) en un temps fini. En exécutant cette règle, *r* amorce la diffusion d'un message *m* : PIF_r passe de C à B, une question est initié ($Que_r := Q$) et la valeur In est affecté à $Request_r$ (pour signifier à l'application que sa requête a été prise en compte).

Ensuite, les autres processeurs reçoivent le message *m* (règle Bst) lorsqu'ils ne participent à aucun autre cycle de PIR (*i.e.*, $PIF = C$) et qu'au moins un de leurs voisins est en train de diffuser le messsage ($PIF = B$). Lorsqu'un processeur p exécute sa règle Bst, il passe dans une phase de diffusion ($PIF_p := B$), initie une *question* par $Que_p := Q$, désigne son père avec Par_p et finalement évalue sa hauteur dans sa variable de niveau L_p. Le processeur p est ensuite supposé diffuser le message *m* à tous ses voisins sauf Par_p. Ainsi, pas à pas, la phase de diffusion progresse dans le réseau (sans attendre les réponses aux questions initiées). En particulier, cette phase construit dynamiquement un arbre couvrant enraciné en *r* (défini par les variables Par).

Toute phase de diffusion initiée par *r* est assurée d'atteindre tous les processeurs du réseau grâce au mécanisme de questions. En fait, à partir du moment où un processeur reçoit un message provenant de *r*, il reste dans la phase de diffusion tant qu'il n'a pas reçu confirmation par la phase de question que tous ses voisins sont dans la phase de diffusion de la racine (*i.e.* tant qu'il ne vérifie pas $CFree$ et $AnswerOk$).

La phase de retour suit la phase de diffusion et est exécutée à partir des feuilles de l'arbre couvrant. Lorsque la diffusion atteint un processeur q tel que tous ses voisins ont déjà reçu le message par d'autres processeurs, q attend la confirmation de la phase de question puis exécute sa règle Fck : il passe de la phase de diffusion à la phase de retour. La phase de retour est ensuite propagée dans l'arbre couvrant construit lors de la diffusion (en suivant les pointeurs pères) comme suit : un processeur interne de l'arbre exécute sa règle Fck quand tous ses voisins ont reçu le message *m* ($CFree$), il y est autorisé par la phase de question ($AnswerOk$) et que tous ses fils dans l'arbre sont dans la phase de retour ($BLeaf$). Ainsi, *r* est le dernier processeur à exécuter la phase de retour (règle Fck). Le cas échéant, *r* initie ensuite une phase de nettoyage (n.b. cette phase est identique à la phase de nettoyage du protocole $\mathcal{DFS}2$). Suite à cette phase de nettoyage, *r* exécute sa règle de terminaison T : *r* affecte Out à $Request_r$ pour signifier à l'application que le PIR demandé est terminé. Après cette dernière affectation, *r* attend la prochaine requête, *i.e.*, le prochain message à envoyer.

Enfin, nous rappelons que la phase de correction permet d'éviter tout interblocage dans le système en supprimant les processus anormaux du système (n.b. la phase de correction reprend les mêmes principes que la phase de correction de $\mathcal{DFS}2$).

Nous allons maintenant rappeler des résultats formels sur le protocole $\mathcal{PIR}$. Ces résultats ont été publiés dans [CDV06b][2]. Ils seront utilisés dans la preuve de la stabilisation instantanée de notre transformateur, $\mathcal{SSBB}$.

[2]Ces résultats sont disponibles sur internet, cf. [CDV06c].

Algorithm 6.2.14 Algorithme $\mathcal{PIR}$ pour $p \neq r$

Entrée : $Neig_p$: ensemble des voisins (localement ordonné) ;

Variables : $PIF_p \in \{B,F,PC,C,EB,EF\}$; $Par_p \in Neig_p$; $L_p \in \mathbb{N}$; $Que_p \in \{Q,R,W,A\}$;

Macros :

$Child_p = \{q \in Neig_p :: (PIF_q \neq C) \wedge (Par_q = p) \wedge (L_q = L_p + 1)$
$\wedge [(PIF_q \neq PIF_p) \Rightarrow ((PIF_p \in \{B,PC\} \wedge PIF_q = F) \vee (PIF_p = EB))]\}$;

$Pre\_Potential_p = \{q \in Neig_p :: PIF_q = B\}$;

$Potential_p = \{q \in Neig_p :: \forall q' \in Pre\_Potential_p, L_q \leq L_{q'}\}$;

Prédicats généraux :

$CFree(p) \equiv (\forall q \in Neig_p :: PIF_q \neq C)$

$Leaf(p) \equiv [\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Par_q \neq p)]$

$BLeaf(p) \equiv (PIF_p = B) \wedge [\forall q \in Neig_p :: (Par_q = p) \Rightarrow (PIF_q = F)]$

$AnswerOk(p) \equiv (Que_p = A) \wedge [\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q = A)]$

$GoodPIF(p) \equiv (PIF_p = C) \vee [(PIF_{Par_p} \neq PIF_p) \Rightarrow ((PIF_{Par_p} = EB)$
$\vee (PIF_p = F \wedge PIF_{Par_p} \in \{B,PC\}))]$

$GoodL(p) \equiv (PIF_p \neq C) \Rightarrow (L_p = L_{Par_p} + 1)$

$AbRoot(p) \equiv \neg GoodPIF(p) \vee \neg GoodL(p)$

Gardes :

$EFAbRoot(p) \equiv (PIF_p = EF) \wedge AbRoot(p) \wedge [\forall q \in Neig_p :: (Par_q{=}p \wedge L_q{>}L_p) \Rightarrow (PIF_q{\in}\{EF,C\})]$

$EBroadcast(p) \equiv (PIF_p \in \{B,F,PC\}) \wedge [\neg AbRoot(p) \Rightarrow (PIF_{Par_p} = EB)]$

$EFeedback(p) \equiv (PIF_p = EB) \wedge [\forall q \in Neig_p :: (Par_q = p \wedge L_q > L_p) \Rightarrow (PIF_q \in \{EF,C\})]$

$Broadcast(p) \equiv (PIF_p = C) \wedge (Potential_p \neq \emptyset) \wedge Leaf(p)$

$Feedback(p) \equiv BLeaf(p) \wedge CFree(p) \wedge AnswerOk(p)$

$PreClean(p) \equiv (PIF_p = F) \wedge (PIF_{Par_p} = PC) \wedge [\forall q \in Neig_p :: (Par_q = p) \Rightarrow (PIF_q \in \{F,C\})]$

$Cleaning(p) \equiv (PIF_p = PC) \wedge Leaf(p)$

$Require(p) \equiv (PIF_p \in \{B,F\}) \wedge [(PIF_p = B) \Rightarrow CFree(p)]$
$\wedge [[(Que_p = Q) \wedge (\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q \in \{Q,R\}))]$
$\vee [(Que_p \in \{W,A\}) \wedge (\exists q \in Neig_p :: (PIF_q \neq C) \wedge ((Que_q = Q)$
$\vee (q \in Child_p \wedge Que_q = R)))]]$

$Wait(p) \equiv (PIF_p \in \{B,F\}) \wedge [(PIF_p = B) \Rightarrow CFree(p)] \wedge (Que_p = R) \wedge (Que_{Par_p} = R)$
$\wedge (\forall q \in Child_p :: Que_q \in \{W,A\}) \wedge (\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q \neq Q))$

$Answer(p) \equiv (PIF_p \in \{B,F\}) \wedge [(PIF_p = B) \Rightarrow CFree(p)] \wedge (Que_p = W) \wedge (Que_{Par_p} = A)$
$\wedge (\forall q \in Child_p :: Que_q \in \{W,A\}) \wedge (\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q \neq Q))$

Règles :

Phase de Correction :

$EC :: EFAbRoot(p) \rightarrow PIF_p := C$;

$EB :: EBroadcast(p) \rightarrow PIF_p := EB$;

$EF :: EFeedback(p) \rightarrow PIF_p := EF$;

Phase de PIR :

$Bst :: Broadcast(p) \rightarrow PIF_p := B$; $Par_p := \min_{\prec_p}(Potential_p)$;
$L_p := L_{Par_p} + 1$; $Que_p := Q$;

$Fck :: Feedback(p) \rightarrow PIF_p := F$;

$PC :: PreClean(p) \rightarrow PIF_p := PC$;

$C :: Cleaning(p) \rightarrow PIF_p := C$;

Phase de Question :

$QR :: Require(p) \rightarrow Que_p := R$;

$QW :: Wait(p) \rightarrow Que_p := W$;

$QA :: Answer(p) \rightarrow Que_p := A$;

6.2.2 Quelques résultats sur le protocole PIR

Théorème 6.2.1 ([CDV06b]) *Le protocole $\mathcal{PIR}$ est un protocole de PIR instantanément stabilisant sous un démon distribué inéquitable.*

Comme le démon distribué inéquitable est le démon le plus général du modèle à états, le théorème 6.2.1 implique que le protocole $\mathcal{PIR}$ fonctionne avec n'importe quel type de démon. Une autre conséquence du théorème 6.2.1 est qu'à partir de n'importe quelle configuration, chaque cycle de PIR est exécuté en un nombre fini de mouvements.

Propriété 6.2.1 *D'après [CDV06c], nous avons les résultats suivants :*

1. *Après que* r *ait initié une diffusion (règle Bst), le système atteint en un temps fini une configuration où chaque processeur est dans la phase de retour associée à cette diffusion.*
2. *À partir de n'importe quelle configuration et suite à une requête,* r *exécute sa règle Bst (*i.e. r *initie la diffusion d'un nouveau message) en $O(\Delta \times n^3)$ mouvements et $O(n)$ rondes.*
3. *À partir de n'importe quelle configuration, un PIR complet est exécuté en $O(\Delta \times n^3)$ mouvements et $O(n)$ rondes.*
4. *$\forall p \in V$, à partir de n'importe quelle configuration où $PIF_p = C$, p est sûr d'accuser réception d'un message initié par* r *dès la 2ème exécution de sa règle Fck.*

Le lemme suivant montre que le pire des cas en temps d'exécution du protocole $\mathcal{PIR}$ correspond à l'exécution du premier cycle de calcul. En effet, les autres cycles sont exécutée en $O(n)$ mouvements :

Théorème 6.2.2 *Après le premier PIR initié, les autres cycles de PIR sont exécutés en $O(n)$ mouvements.*

Preuve. D'après le point 1 de la propriété 6.2.1, nous savons que tous les comportements anormaux dûs à la configuration initiale sont corrigés durant l'exécution du premier PIR initié. Donc, suite à ce premier PIR, les autres cycles d'exécution démarrent toujours dans une configuration où $\forall p \in V$, $PIF_p = C$. À partir d'une telle configuration, les règles de la phase de correction ne sont plus jamais exécutées. D'autre part, chaque cycle initié comporte trivialement $O(n)$ actions des phases de PIR et de nettoyage.

Focalisons nous maintenant sur la phase de question. Cette phase comporte trois règles : QR, QW et QA. Ces règles peuvent être exécutées uniquement par un processeur participant au PIR initié par *r* (n.b. il n'existe plus de PIR anormaux dans le système), *i.e.*, un processeur tel que $PIF \neq C$. Un processeur p participe au PIR initié par la racine à partir du moment où il exécute la règle Bst. Par cette règle, p affecte la valeur Q à la variable Que_p. Ensuite, les règles QR, QW et QA permettent à tout processeur $p \neq$ *r* d'affecter successivement les valeurs R, W et A à la variable Que_p. D'autre part, les règles QR et QA de *r* (n.b. *r* n'a pas de règle QW) lui permettent d'affecter successivement les valeurs R et A à la variable Que_r.

Nous pouvons alors remarquer que, grâce au prédicat $Wait(p)$, $p \neq$ *r* passe Que_p de R à W uniquement lorsque la construction de son sous-arbre de diffusion est complètement terminée et que tous ses descendants dans l'arbre vérifie $Que = W$. D'après le prédicat $Require(p)$, p ne peut donc plus exécuter la règle QR dans le cycle de PIR courant. De la même manière, grâce au prédicat $Answer(p)$, r passe Que_r de R à A uniquement lorsque la construction de l'arbre de diffusion est complètement terminée et que tous les autres processeurs vérifient $Que = W$. Donc, d'après le prédicat $Require(r)$, *r* ne peut plus exécuter la règle QR dans le cycle de PIR courant.

Or, pour $p \neq$ *r*, chaque règle QA est exécutée après l'exécution d'une règle QW qui suit elle-même l'exécution d'une règle QR. Similairement, *r* exécute une règle QA pour chaque règle QR.

Algorithm 6.3.15 Algorithme $Reset(\mathcal{B})$ pour $p = r$

Entrée : $Neig_p$: ensemble des voisins (localement ordonné) ;

Constantes : $Par_p = \perp$; $L_p = 0$;

Variables : $PIF_p \in \{B,F,PC,C\}$; $Que_p \in \{Q,R,A\}$;

Macro :

$Child_p = \{q \in Neig_p :: (PIF_q \neq C) \wedge (Par_q = p) \wedge (L_q = L_p + 1) \wedge [(PIF_q \neq PIF_p) \Rightarrow (PIF_p \in \{B,PC\} \wedge PIF_q = F)]\}$;

Prédicats généraux :

$CFree(p) \equiv (\forall q \in Neig_p :: PIF_q \neq C)$

$Leaf(p) \equiv [\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Par_q \neq p)]$

$BLeaf(p) \equiv (PIF_p = B) \wedge [\forall q \in Neig_p :: (Par_q = p) \Rightarrow (PIF_q = F)]$

$AnswerOk(p) \equiv (Que_p = A) \wedge [\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q = A)]$

Gardes :

$Broadcast(p) \equiv (PIF_p = C) \wedge Leaf(p)$

$Feedback(p) \equiv BLeaf(p) \wedge CFree(p) \wedge AnswerOk(p)$

$PreClean(p) \equiv (PIF_p = F) \wedge [\forall q \in Neig_p :: (Par_q = p) \Rightarrow (PIF_q \in \{F,C\})]$

$Cleaning(p) \equiv (PIF_p = PC) \wedge Leaf(p)$

$Require(p) \equiv (PIF_p \in \{B,F\}) \wedge [(PIF_p = B) \Rightarrow CFree(p)] \wedge [[(Que_p = Q) \wedge (\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q \in \{Q,R\}))] \vee [(Que_p{=}A) \wedge (\exists q \in Neig_p :: (PIF_q{\neq}C) \wedge ((Que_q{=}Q) \vee (q{\in}Child_p \wedge Que_q{=}R)))]]$

$Answer(p) \equiv (PIF_p{\in}\{B,F\}) \wedge [(PIF_p{=}B){\Rightarrow}CFree(p)] \wedge (Que_p{=}R) \wedge (\forall q{\in}Child_p{::}Que_q{\in}\{W,A\}) \wedge [\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q \neq Q)]$

Règles :

Phase de PIR :

$Bst :: (\mathcal{B}.Request_r{=}Wait){\wedge}\mathcal{B}.End(p){\wedge}Broadcast(p) \rightarrow PIF_p{:=}B ; Que_p{:=}Q ;$ /* Démarrage */ $\mathcal{B}.Request_r := In ;$

$Fck :: Feedback(p) \rightarrow PIF_p := F ; \mathcal{B}.Init_p ;$

$PC :: PreClean(p) \rightarrow PIF_p := PC ;$

$C :: Cleaning(p) \rightarrow PIF_p := C ;$

$T :: (\mathcal{B}.Request_r = In) \wedge \mathcal{B}.End(p) \wedge (PIF_p = C) \rightarrow \mathcal{B}.Request_r{:=}Out ;$ /* Terminaison */

Phase de Question :

$QR :: Require(p) \rightarrow Que_p := R ;$

$QA :: Answer(p) \rightarrow Que_p := A ;$

Donc, chaque processeur exécute une fois et une seule chacune de ses règles de la phase de question pour chaque cycle de PIR initié à partir d'une configuration où $\forall p \in V$, $PIF_p = C$. D'où, après le premier cycle de PIR initié, les autres cycles comportent $O(n)$ actions de la phase de question et le théorème est vérifié. □

6.3 Transformateur

Nous allons maintenant expliquer comment construire notre transformateur, $\mathcal{SSBB}$ en utilisant le protocole $\mathcal{PIR}$.

6.3.1 Le protocole

Pour construire notre transformateur $\mathcal{SSBB}$, nous utilisons la technique de composition suivante. Soient P_1 et P_2 deux protocoles. La composition de P_1 et P_2, notée $P_2 \oplus_{|G} P_1$, est le protocole vérifiant les conditions suivantes :

1. $P_2 \oplus_{|G} P_1$ contient toutes les variables et règles de P_1 et P_2.
2. G est un prédicat défini sur les variables de P_1.

Algorithm 6.3.16 Algorithme $Reset(\mathcal{B})$ pour $p \neq r$

Entrée : $Neig_p$: ensemble des voisins (localement ordonné) ;

Variables : $PIF_p \in \{B,F,PC,C,EB,EF\}$; $Par_p \in Neig_p$; $L_p \in \mathbb{N}$; $Que_p \in \{Q,R,W,A\}$;

Macros :

$Child_p = \{q \in Neig_p :: (PIF_q \neq C) \wedge (Par_q = p) \wedge (L_q = L_p + 1) \wedge [(PIF_q \neq PIF_p) \Rightarrow ((PIF_p \in \{B,PC\} \wedge PIF_q = F) \vee (PIF_p = EB))]\}$;

$Pre\_Potential_p = \{q \in Neig_p :: PIF_q = B\}$;

$Potential_p = \{q \in Neig_p :: \forall q' \in Pre\_Potential_p, L_q \leq L_{q'}\}$;

Prédicats généraux :

$CFree(p) \equiv (\forall q \in Neig_p :: PIF_q \neq C)$

$Leaf(p) \equiv [\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Par_q \neq p)]$

$BLeaf(p) \equiv (PIF_p = B) \wedge [\forall q \in Neig_p :: (Par_q = p) \Rightarrow (PIF_q = F)]$

$AnswerOk(p) \equiv (Que_p = A) \wedge [\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q = A)]$

$GoodPIF(p) \equiv (PIF_p{=}C) \vee [(PIF_{Par_p} \neq PIF_p) \Rightarrow ((PIF_{Par_p}{=}EB) \vee (PIF_p{=}F \wedge PIF_{Par_p} \in \{B,PC\}))]$

$GoodL(p) \equiv (PIF_p \neq C) \Rightarrow (L_p = L_{Par_p} + 1)$

$AbRoot(p) \equiv \neg GoodPIF(p) \vee \neg GoodL(p)$

Gardes :

$EFAbRoot(p) \equiv (PIF_p{=}EF) \wedge AbRoot(p) \wedge [\forall q{\in}Neig_p :: (Par_q{=}p \wedge L_q{>}L_p) \Rightarrow (PIF_q{\in}\{EF,C\})]$

$EBroadcast(p) \equiv (PIF_p \in \{B,F,PC\}) \wedge [\neg AbRoot(p) \Rightarrow (PIF_{Par_p} = EB)]$

$EFeedback(p) \equiv (PIF_p = EB) \wedge [\forall q \in Neig_p :: (Par_q = p \wedge L_q > L_p) \Rightarrow (PIF_q \in \{EF,C\})]$

$Broadcast(p) \equiv (PIF_p = C) \wedge (Potential_p \neq \emptyset) \wedge Leaf(p)$

$Feedback(p) \equiv BLeaf(p) \wedge CFree(p) \wedge AnswerOk(p)$

$PreClean(p) \equiv (PIF_p = F) \wedge (PIF_{Par_p} = PC) \wedge [\forall q \in Neig_p :: (Par_q = p) \Rightarrow (PIF_q \in \{F,C\})]$

$Cleaning(p) \equiv (PIF_p = PC) \wedge Leaf(p)$

$Require(p) \equiv (PIF_p \in \{B,F\}) \wedge [(PIF_p = B) \Rightarrow CFree(p)] \wedge [[(Que_p = Q) \wedge (\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q \in \{Q,R\}))] \vee [(Que_p{\in}\{W,A\}) \wedge (\exists q{\in}Neig_p :: (PIF_q{\neq}C) \wedge ((Que_q{=}Q){\vee}(q{\in}Child_p{\wedge}Que_q{=}R)))]]$

$Wait(p) \equiv (PIF_p \in \{B,F\}) \wedge [(PIF_p = B) \Rightarrow CFree(p)] \wedge (Que_p = R) \wedge (Que_{Par_p} = R) \wedge (\forall q \in Child_p :: Que_q \in \{W,A\}) \wedge (\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q \neq Q))$

$Answer(p) \equiv (PIF_p \in \{B,F\}) \wedge [(PIF_p = B) \Rightarrow CFree(p)] \wedge (Que_p = W) \wedge (Que_{Par_p} = A) \wedge (\forall q \in Child_p :: Que_q \in \{W,A\}) \wedge (\forall q \in Neig_p :: (PIF_q \neq C) \Rightarrow (Que_q \neq Q))$

Règles :

Phase de Correction :

$EC :: EFAbRoot(p) \rightarrow PIF_p := C$;

$EB :: EBroadcast(p) \rightarrow PIF_p := EB$;

$EF :: EFeedback(p) \rightarrow PIF_p := EF$;

Phase de PIR :

$Bst :: Broadcast(p) \rightarrow PIF_p := B$; $Par_p := \min_{\prec_p}(Potential_p)$; $L_p{:=}L_{Par_p}{+}1$; $Que_p{:=}Q$;

$F :: Feedback(p) \rightarrow PIF_p := F$; $\mathcal{B}.Init_p$;

$PC :: PreClean(p) \rightarrow PIF_p := PC$;

$C :: Cleaning(p) \rightarrow PIF_p := C$;

Phase de Question :

$QR :: Require(p) \rightarrow Que_p := R$;

$QW :: Wait(p) \rightarrow Que_p := W$;

$QA :: Answer(p) \rightarrow Que_p := A$;

3. Chaque règle $L_i :: H_i \rightarrow S_i$ de P_2 est remplacée dans le protocole $P_2 \oplus_{|G} P_1$ par $L_i :: G \wedge H_i \rightarrow S_i$.

En suivant ces règles de composition, $\mathcal{SSBB}(\mathcal{B}) = \mathcal{B} \oplus_{|Ok(p)} Reset(\mathcal{B})$ où $Ok(p) \equiv (PIF_p = C)$. $Reset(\mathcal{B})$ (cf. Algorithmes 6.3.15 et 6.3.16) est une version légèrement modifiée du protocole $\mathcal{PIR}$. $Reset(\mathcal{B})$ est utilisé pour réinitialiser les variables du protocole $\mathcal{B}$ avant toute exécution de $\mathcal{B}$. Nous avons ainsi remplacé la variable $Request_r$ par $\mathcal{B}.Request_r$ dans le code de $Reset(\mathcal{B})$ pour signifier que $Reset(\mathcal{B})$ prend en compte les requêtes pour $\mathcal{B}$ à la place de $\mathcal{B}$. Puis, nous avons modifié l'action de démarrage de $\mathcal{SSBB}(\mathcal{B})$ (la règle Bst de *r* dans $Reset(\mathcal{B})$) pour que l'action de démarrage soit activable seulement si $\mathcal{B}.End(r)$ est vérifiée (afin d'empêcher l'abandon d'un calcul précédemment initié de la tâche $\mathcal{TASK}$). Ensuite, nous avons modifié la règle Fck pour que les variables de $\mathcal{B}$ soient réinitialisées en utilisant $\mathcal{B}.Init_p$ durant la phase de retour. Finalement, nous utilisons le prédicat $Ok(p)$ dans la composition pour que chaque processeur p stoppe son exécution locale de $\mathcal{B}$ à partir du moment où il reçoit le message d'abandon (la règle Bst) et jusqu'à la terminaison locale en p du cycle de réinitialisation (jusqu'à ce que $S_p := C$). En effet, nous savons déjà que p satisfait continûment $PIF_p \neq C$ durant sa participation à une réinitialisation globale du système (cf. section précédente). Donc, tant que p participe à une réinitialisation, $Ok(p)$ est faux et aucune règle de $\mathcal{B}$ dans $\mathcal{SSBB}(\mathcal{B})$ n'est activable en p.

6.3.2 Preuve de la stabilisation instantanée

Nous allons maintenant montrer que le protocole composite $\mathcal{SSBB}(\mathcal{B})$ (où $\mathcal{B}$ est un protocole vérifiant $SSBB\text{-}Friendly(\mathcal{TASK},\mathcal{D})$) est instantanément stabilisant pour les spécifications de la tâche $\mathcal{TASK}$ (la tâche résolue par $\mathcal{A}$ dans un environnement sans faute) sous un démon $\mathcal{D}$. Contrairement aux protocoles précédents, nous allons explicitement faire référence à la variable $Request_r$ dans les preuves suivantes.

Tout d'abord, comme $\mathcal{B}$ est uniquement écrit pour résoudre la tâche spécifique $\mathcal{TASK}$ en fournissant le prédicat $\mathcal{B}.End(r)$ et les macros $\mathcal{B}.Init_p$ indépendamment des variables de $Reset(\mathcal{B})$, nous pouvons poser l'hypothèse suivante :

Hypothèse 6.3.1 *$\mathcal{B}$ n'écrit pas dans les variables de $Reset(\mathcal{B})$.*

Nous allons maintenant prouver que $\mathcal{SSBB}(\mathcal{B})$ est une *composition équitable* des protocoles $\mathcal{B}$ et $Reset(\mathcal{B})$ (Définition 5.7.2, page 100).

Théorème 6.3.1 *$\mathcal{SSBB}(\mathcal{B})$ est une* composition équitable *des protocoles $\mathcal{B}$ et $Reset(\mathcal{B})$.*

Preuve. Supposons, par contradiction, qu'il existe au moins une exécution e de $\mathcal{SSBB}(\mathcal{B})$ qui ne soit pas *équitable*. Dans ce cas, d'après la définition 5.7.2 (page 100), e est infinie et deux cas sont possibles :

1. Il existe un suffixe infini e' de e où des règles de $\mathcal{B}$ sont activables mais seules des règles de $Reset(\mathcal{B})$ sont exécutées. Alors, des règles de $Reset(\mathcal{B})$ sont exécutées infiniment souvent dans e'. Or, d'après la propriété 6.2.1 (point 1), chaque cycle de calcul de $Reset(\mathcal{B})$ est exécuté en un nombre fini de mouvements. Donc, $Reset(\mathcal{B})$ exécute des cycles de calcul infiniment souvent dans e'. D'après l'hypothèse 6.3.1, les variables de $Reset(\mathcal{B})$ se comportent dans e' comme le protocole $\mathcal{PIR}$ qui est instantanément stabilisant d'après le théorème 6.2.1. Donc, durant le premier cycle complet de calcul dans e', $Reset(\mathcal{B})$ exécute, en particulier, une phase de retour complète. Or, quand *r* exécute sa phase de retour (règle Fck), *r* réinitialise ses variables de $\mathcal{B}$ en utilisant $\mathcal{B}.Init_r$. Après cette réinitialisation locale, $\mathcal{B}.End(r)$ devient faux et,

comme les variables de $\mathcal{B}$ ne sont plus jamais modifiées (par hypothèse, aucune règle de $\mathcal{B}$ n'est exécutée dans e'), $\mathcal{B}.End(r)$ est faux pour toujours. Cela implique que l'action de démarrage de $Reset(\mathcal{B})$, *i.e.*, la règle Bst de *r*, devient inactivable pour toujours. D'où, $Reset(\mathcal{B})$ ne peut pas exécuter des cycles de calcul infiniment souvent dans e', contradiction.

2. Il existe un suffixe infini e' de e où des règles de $Reset(\mathcal{B})$ sont activables mais seules des règles de $\mathcal{B}$ sont exécutées. En particulier, cela signifie que des règles de $\mathcal{B}$ sont exécutées infiniment souvent dans e'.

   En fonction des règles de $Reset(\mathcal{B})$ activables dans e', nous étudions maintenant les cas suivants :
   - Supposons que la règle Bst de *r* est activable dans une configuration γ_i telle que $\gamma_i \in e'$. Alors, $(\mathcal{B}.Request_r = Wait) \wedge \mathcal{B}.End(r) \wedge Broadcast(r)$ est vérifié dans γ_i. Comme $\mathcal{B}.Request_r = Wait$ jusqu'à ce que *r* exécute sa règle Bst, $\mathcal{B}.Request_r$ est continûment égale à $Wait$ à partir de γ_i. Ensuite, comme aucune règle de $Reset(\mathcal{B})$ n'est exécutée dans e' et $\mathcal{B}$ n'écrit pas dans les variables de $Reset(\mathcal{B})$ (hypothèse 6.3.1), $Broadcast(r)$ est aussi continûment vérifiée à partir de γ_i. D'où, la valeur de la garde de la règle Bst dépend maintenant de $\mathcal{B}.End(r)$ uniquement. Or, comme $\mathcal{B}.End(r)$ est vérifiée dans γ_i et que $\mathcal{B}.End(r)$ est *stable* pour $\mathcal{D}$, d'après la définition 6.1.1, la règle Bst est exécutée en un temps fini en dépit de $\mathcal{D}$ (le démon), contradiction.
   - Supposons que la règle T de *r* est activable dans une configuration γ_i telle que $\gamma_i \in e'$. Par un raisonnement similaire à celui utilisé dans le cas précédent, nous obtenons aussi une contradiction.
   - Supposons que des règles de $Reset(\mathcal{B})$ différentes des règles Bst et T de *r* sont activables dans une configuration γ_i telle que $\gamma_i \in e'$. De telles règles dépendent de variables internes de $Reset(\mathcal{B})$ uniquement. Donc, comme $\mathcal{B}$ n'écrit pas dans les variables de $Reset(\mathcal{B})$ (hypothèse 6.3.1), ces règles sont continûment activables et, par contradiction, $\mathcal{D} = DUF$: seul un démon inéquitable peut empêcher une règle continûment activable d'être exécutée. Or, la conséquence 6.1.1 (point 3) implique que le système atteint en un temps fini une configuration γ_j où seules les règles de $Reset(\mathcal{B})$ sont activables dans $\mathcal{SSBB}(\mathcal{B})$ (n.b. $\mathcal{B}.End(r)$ est *stable* pour un démon distribué inéquitable). D'où, au moins une règle activable de $Reset(\mathcal{B})$ est exécutée dans $\gamma_j \mapsto \gamma_{j+1}$, contradiction.

Ainsi, $\mathcal{SSBB}(\mathcal{B})$ est bien une composition équitable des protocoles $\mathcal{B}$ et $Reset(\mathcal{B})$. □

Le lemme suivant montre qu'à partir de toute configuration où le système est en attente d'un calcul de la tâche $\mathcal{TASK}$ ($\mathcal{B}.Request_r = Wait$), $\mathcal{SSBB}(\mathcal{B})$ est initié en un temps fini.

Lemme 6.3.1 *À partir de toute configuration où $\mathcal{B}.Request_r = Wait$, $\mathcal{SSBB}(\mathcal{B})$ est initié en un temps fini.*

Preuve. Supposons, par contradiction, qu'à partir de toute configuration où $\mathcal{B}.Request_r = Wait$, *r* n'exécute jamais la règle Bst (*i.e.*, l'action de démarrage de $\mathcal{SSBB}(\mathcal{B})$). Tout d'abord, la règle Bst de *r* est la seule règle qui peut modifier $\mathcal{B}.Request_r$ quand $\mathcal{B}.Request_r = Wait$ (cf. algorithme 6.3.15 et la définition 6.1.3, point 2). Donc, *r* vérifie $\mathcal{B}.Request_r = Wait$ pour toujours. Ensuite, d'après le théorème 6.3.1 et l'hypothèse 6.3.1, $\mathcal{B}$ ne perturbe pas le comportement de $Reset(\mathcal{B})$. Donc, d'après la propriété 6.2.1 (point 2), en dépit du démon $\mathcal{D}$, le système atteint en un nombre fini de pas une configuration où aucune règle de $Reset(\mathcal{B})$ n'est activable et *r* vérifie $Broadcast(r)$ pour toujours. À partir de cette configuration, la valeur de la garde de la règle Bst de *r* dépend uniquement de $\mathcal{B}.End(r)$. Or, en dépit de $\mathcal{D}$, *r* finit par vérifier $\mathcal{B}.End(r)$ car $\mathcal{B}.End(r)$ satisfait $BreakingIn(\mathcal{D})$ (cf. définition 6.1.2) et, comme $\mathcal{B}.End(r)$ est stable pour $\mathcal{D}$, *r* finit par exécuter la règle Bst en dépit du démon $\mathcal{D}$ d'après la définition 6.1.1, contradiction. □

Le lemme suivant montre que suite à une demande de calcul de la tâche $\mathcal{TASK}$ par une application (formellement, dès que le prédicat $ApplicationRequest(r)$ de la règle externe IR est vérifié), le système prend cette demande en compte en un temps fini en exécutant $\mathcal{B}.Request_r := Wait$.

Lemme 6.3.2 *À partir d'une configuration où un calcul de $\mathcal{SSBB}(\mathcal{B})$ est demandé,* r *exécute la règle IR en un temps fini.*

Preuve. Supposons, par contradiction, qu'à partir d'une configuration γ_i un calcul de $\mathcal{SSBB}(\mathcal{B})$ soit demandé à *r* mais que *r* n'exécute jamais la règle IR (*i.e.*, la règle externe contenant l'affectation $\mathcal{B}.Request_r := Wait$).

- Supposons $\mathcal{B}.Request_r = Out$ dans γ_i. Dans une telle configuration, la règle IR de *r* est activable (n.b. la garde de IR est la suivante : $ApplicationRequest(r) \wedge (Request_r = Out)$). De plus, aucune règle de $\mathcal{SSBB}(\mathcal{B})$ ne peut modifier $\mathcal{B}.Request_r$ jusqu'à ce que *r* affecte $Wait$ à $\mathcal{B}.Request_r$ par IR. Donc, la règle IR de *r* est continûment activable et si $\mathcal{D}\in\{DSF,DWF\}$, alors IR est exécutée en un temps fini, contradiction. D'où, par contradiction, $\mathcal{D} = DUF$. Comme IR n'est jamais exécutée, $\mathcal{B}.Request_r$ reste égale à Out pour toujours. En particulier, cela signifie que la règle Bst de *r* n'est plus jamais activable à partir de γ_i. Comme, d'après le théorème 5.4.4 et l'hypothèse 6.3.1, $\mathcal{B}$ ne perturbe pas le comportement du protocole $Reset(\mathcal{B})$, la propriété 6.2.1 (point 2) implique qu'en dépit du démon $\mathcal{D}$, le système finit par atteindre une configuration γ_j à partir de laquelle aucune règle de $Reset(\mathcal{B})$ n'est plus jamais activable (n.b. *r* n'exécute plus jamais la règle Bst). À partir γ_j, seules les règles de $\mathcal{B}$ sont exécutées. Or, dans ce cas, le système converge en temps fini de γ_j à une configuration γ_k où la règle IR est la seule règle activable du système d'après la conséquence 6.1.1 (point 3). D'où, IR est exécutée dans $\gamma_k \mapsto \gamma_{k+1}$, contradiction.
- Supposons que $\mathcal{B}.Request_r = In$ dans γ_i. À partir de γ_i, $\mathcal{B}.Request_r$ reste égal à In jusqu'à ce que *r* exécute la règle T (cf. algorithmes 5.4.3, 5.4.4 et définition 6.1.3 point 2). Supposons alors que *r* finit par exécuter la règle T. Par la règle T, $\mathcal{B}.Request_r := Out$ et, d'après le cas précédent, *r* finit par exécuter la règle IR, contradiction. Supposons donc, par contradiction, que *r* n'exécute jamais la règle T. $\mathcal{B}.Request_r$ reste donc égale à In pour toujours. En particulier, cela signifie que la règle Bst de *r* n'est plus jamais activable à partir de γ_i. Comme, d'après le théorème 5.4.4 et l'hypothèse 6.3.1, $\mathcal{B}$ ne perturbe pas le comportement du protocole $Reset(\mathcal{B})$, la propriété 6.2.1 (point 2) implique qu'en dépit du démon $\mathcal{D}$, le système finit par atteindre une configuration γ_j à partir de laquelle aucune règle de $Reset(\mathcal{B})$ n'est plus jamais activable et *r* vérifie $Broadcast(r)$ pour toujours (n.b. *r* n'exécute plus jamais la règle Bst). À partir de γ_j, $\mathcal{B}.Request_r = In \wedge PIF_r = C$ pour toujours ($Broadcast(r) \Rightarrow (PIF_r = C)$) et la valeur de la garde de la règle T de *r* dépend maintenant uniquement de $\mathcal{B}.End(p)$. Or, en dépit de la nature de $\mathcal{D}$, *r* finit par vérifier $\mathcal{B}.End(p)$ car $\mathcal{B}.End(p)$ vérifie $BreakingIn(\mathcal{D})$ (cf. définition 6.1.2) et, comme $\mathcal{B}.End(p)$ est ***stable*** pour $\mathcal{D}$, *r* finit par exécuter T en dépit de $\mathcal{D}$ d'après la définition 6.1.1, contradiction.
- Supposons que $\mathcal{B}.Request_r = Wait$ dans γ_i. Alors, d'après le lemme 6.3.1, nous savons que *r* finit par exécuter la règle Bst. Par la règle Bst, $\mathcal{B}.Request_r := In$ et nous retrouvons le cas précédent, contradiction.

□

D'après les lemmes 6.3.1 et 6.3.2, nous déduisons le théorème suivant. Ce théorème signifie que suite à une demande de calcul de la tâche $\mathcal{TASK}$, $\mathcal{SSBB}(\mathcal{B})$ est initié en un temps fini.

Théorème 6.3.2 *À partir d'une configuration où un calcul de $\mathcal{SSBB}(\mathcal{B})$ est demandé, le calcul demandé est initié en un temps fini.*

Le théorème suivant montre que chaque calcul de la tâche $\mathcal{TASK}$ initié par *r* est exécuté conformément à ses spécifications.

Théorème 6.3.3 *À partir d'une configuration où* r *initie un calcul de $\mathcal{SSBB}(\mathcal{B})$, la tâche $\mathcal{TASK}$ est exécutée conformément à ses spécifications.*

Preuve. $\mathcal{SSBB}(\mathcal{B})$ démarre par une exécution de $Reset(\mathcal{B})$. Ce protocole est initié par la règle Bst de *r*. Par la règle Bst, *r* exécute, en particulier, $PIF_r := B$. $Reset(\mathcal{B})$ termine ensuite quand *r* affecte C à PIF_r via sa règle C. Donc, durant tout le calcul de $Reset(\mathcal{B})$, *r* vérifie sans discontinuer $PIF_p \neq C$, i.e., $\neg Ok(r)$, et *r* ne peut exécuter aucune règle de $\mathcal{B}$. En particulier, *r* ne peut pas initier $\mathcal{B}$ avant que $Reset(\mathcal{B})$ termine (en *r*).

D'après le théorème 5.4.4 et l'hypothèse 6.3.1, $\mathcal{B}$ ne perturbe pas le comportement de $Reset(\mathcal{B})$. Or, d'après la propriété 6.2.1 (Point 1), après que *r* ait initié une diffusion (règle Bst), le système atteint en un temps fini une configuration γ_i où chaque processeur du système est dans la phase de retour associée à la diffusion initiée par *r*. Donc, comme les variables de $\mathcal{B}$ sont réinitialisées localement en p quand p exécute $PIF_p := F$ et qu'aucune règle de $\mathcal{B}$ n'est exécutée par p tant que $PIF_p \neq C$ ($(PIF_p \neq C) \Rightarrow \neg Ok(p)$), la configuration γ_i restreinte aux variables de $\mathcal{B}$ correspond à la configuration générée par $\mathcal{B}.Init$ (*i.e.*, la configuration initiale normale de $\mathcal{A}$). De plus, il faut noter que $\mathcal{B}.End(r)$ devient faux dès que *r* exécute sa règle Fck (à cause de $\mathcal{B}.Init_r$). Le fait que le système atteigne la configuration γ_i induit deux autres conséquences :

1. À partir de γ_i, aucun processeur n'exécute une règle de $\mathcal{B}$ avant que *r* initie $\mathcal{B}$.
2. À partir de γ_i, le système ne contient plus aucun comportement anormal relatif au protocole $Reset(\mathcal{B})$.

D'où, quand $Reset(\mathcal{B})$ termine en *r* par la règle C (*i.e.*, $PIF_r := C$), le système est dans une configuration γ_j où $\forall p \in V$, $PIF_p = C$ (une configuration initiale normale de $Reset(\mathcal{B})$), les variables de $\mathcal{B}$ décrivent la configuration générée par $\mathcal{B}.Init$, et $\mathcal{B}.End(r) = faux$. À partir de γ_j, aucune autre réinitialisation (*i.e.*, le protocole $Reset(\mathcal{B})$) ne sera de nouveau initiée avant que $\mathcal{B}.End(r)$ soit à nouveau vérifié (cf. la garde de la règle Bst), *i.e.*, avant la décision associée à l'exécution de $\mathcal{B}$ que *r* va initier. Donc, à partir de γ_j et jusqu'à ce que $\mathcal{B}.End(r)$, $\forall p \in V$, $PIF_p = C$ et seul le protocole $\mathcal{B}$ fonctionne. D'où, d'après le point 3 de la définition 6.1.3, à partir de γ_j, $\mathcal{B}$ exécute la tâche spécifique $\mathcal{TASK}$ et, quand *r* décide en exécutant la règle T (T sera exécutée quand $PIF_r = C \wedge \mathcal{B}.End(r)$), la tâche $\mathcal{TASK}$ aura été exécutée conformément à ses spécifications. □

Nous déduisons le théorème suivant d'après la remarque 3.2.1 et les théorèmes 6.3.2 et 6.3.3 :

Théorème 6.3.4 *$\mathcal{SSBB}(\mathcal{B})$ est instantanément stabilisant pour les spécifications de la tâche $\mathcal{TASK}$ sous le démon $\mathcal{D}$.*

6.3.3 Complexité

Complexité en espace. Soit $\mathcal{M}(\mathcal{B})$ l'occupation mémoire du protocole $\mathcal{B}$. Dans les algorithmes 6.2.13 et 6.2.14, les valeurs de la variable L ne sont pas bornées. Cependant, nous pouvons remarquer que le protocole $Reset(\mathcal{B})$ reste valide si nous bornons L par n. Donc, chaque processeur a besoin de $\log n$ bits pour stocker sa variable L et en tenant compte des autres variables déclarées dans les algorithmes 6.2.13 et 6.2.14 ainsi que de la complexité en espace de $\mathcal{B}$, nous pouvons déduire le théorème suivant :

Théorème 6.3.5 *L'occupation mémoire de $\mathcal{SSBB}(\mathcal{B})$ est en $O(\log(n) + \mathcal{M}(\mathcal{B}))$ bits par processeur.*

Complexité en temps. La complexité en temps de $\mathcal{SSBB}(\mathcal{B})$ dépend à la fois de $Reset(\mathcal{B})$ et de $\mathcal{B}$. Évidemment, le protocole $Reset(\mathcal{B})$ a la même complexité que le protocole $\mathcal{PIR}$. Ensuite, comme le protocole $\mathcal{B}$ peut être n'importe quel protocole vérifiant $SSBB\text{-}Friendly(\mathcal{TASK},\mathcal{D})$, il faut noter que la complexité globale de $\mathcal{SSBB}(\mathcal{B})$ varie en fonction de l'instance de $\mathcal{B}$ que nous utilisons.

Nous évaluons tout d'abord la complexité en temps de $\mathcal{SSBB}(\mathcal{B})$ en terme de rondes. À cet effet, nous introduisons les notations suivantes sur la complexité en rondes de $\mathcal{B}$: soit $R_1(\mathcal{B})$ le nombre de rondes de $\mathcal{B}$ nécessaires, à partir de n'importe quelle configuration (de $\mathcal{B}$), pour que $\mathcal{B}.End(r)$ soit continûment activable (n.b. ce nombre ne peut pas être borné si $\mathcal{B}$ ne fonctionne qu'avec un démon fortement équitable), soit $R_2(\mathcal{B})$ le nombre de rondes dont $\mathcal{B}$ a besoin pour exécuter la tâche $\mathcal{TASK}$ à partir de la configuration générée par $\mathcal{B}.Init$.

Le théorème suivant nous donne le délai en nombre de rondes du protocole $\mathcal{SSBB}(\mathcal{B})$.

Théorème 6.3.6 *À partir de toute configuration où un calcul de $\mathcal{SSBB}(\mathcal{B})$ est demandé, le calcul demandé est initié en $O(n + R_1(\mathcal{B}) + R_2(\mathcal{B}))$ rondes.*

Preuve. Comme nous l'avons vu dans la preuve du théorème 5.6.2 (page 98), la gestion explicite des requêtes provoque, dans le pire des cas, l'exécution d'un cycle de calcul complet non demandé avant que le système puisse initier un cycle de calcul demandé. Ce cycle de calcul non demandé, comporte une réinitialisation complète exécutée par $Reset(\mathcal{B})$. Cette réinitialisation coûte $O(n)$ rondes d'après la propriété 6.2.1 (point 3) auxquels il faut ajouter un surcoût de $O(R_1(\mathcal{B}) + R_2(\mathcal{B}))$ rondes, en effet :

- Une fois que le système a atteint une configuration où $Reset(\mathcal{B})$ est prêt à démarrer, il faut au plus $R_1(\mathcal{B})$ rondes pour que $\mathcal{B}.End(r)$ devienne continûment activable et ainsi débloquer l'action de démarrage de $Reset(\mathcal{B})$.
- Après que $Reset(\mathcal{B})$ ait réinitialisé les variables de $\mathcal{B}$, il faut que $\mathcal{B}$ exécute la tâche $\mathcal{TASK}$ à partir de la configuration générée par $\mathcal{B}.Init$ ($R_2(\mathcal{B})$ rondes) pour que $\mathcal{B}.End(r)$ soit à nouveau satisfait et que le système puisse à nouveau démarrer.

D'où, un cycle de calcul complet non demandé de $\mathcal{SSBB}(\mathcal{B})$ est exécuté en $O(R_1(\mathcal{B}) + R_2(\mathcal{B}))$ rondes et le théorème est vérifié. □

Le pire des cas du délai correspond au temps de calcul entre le démarrage d'un calcul non demandé et le démarrage "réel" du calcul demandé. Entre ces deux démarrages, un calcul complet mais non demandé est exécuté. Donc le délai et le temps d'exécution d'un calcul de $\mathcal{SSBB}(\mathcal{B})$ sont du même ordre et nous avons trivialement le corollaire suivant :

Corollaire 6.3.1 *À partir de n'importe quelle configuration, un calcul demandé (et complet) de $\mathcal{SSBB}(\mathcal{B})$ est exécuté en $O(n + R_1(\mathcal{B}) + R_2(\mathcal{B}))$ rondes.*

Nous évaluons maintenant la complexité en nombre de mouvements de $\mathcal{SSBB}(\mathcal{B})$. À cet effet, nous introduisons les notations suivantes : soit $S_1(\mathcal{B})$ le nombre maximal de règles que $\mathcal{B}$ exécute avant d'atteindre une configuration terminale (relative aux variables de $\mathcal{B}$), soit $S_2(\mathcal{B})$ le nombre de mouvements dont $\mathcal{B}$ a besoin pour exécuter la tâche $\mathcal{TASK}$ à partir de la configuration générée par $\mathcal{B}.Init$.

Théorème 6.3.7 *À partir de toute configuration où un calcul de $\mathcal{SSBB}(\mathcal{B})$ est demandé, le calcul demandé est initié en $O(\Delta \times n^3 \times S_1(\mathcal{B}))$ mouvements.*

Preuve. Dans la preuve du théorème 6.3.6, nous avons vu que, dans le pire des cas, un cycle de calcul demandé de $\mathcal{SSBB}(\mathcal{B})$ est initié après un cycle de calcul non demandé (complet) de $\mathcal{SSBB}(\mathcal{B})$. D'après la propriété 6.2.1 (point 3), nous savons que ce cycle non demandé contient l'exécution de $O(\Delta \times n^3)$ règles de $Reset(\mathcal{B})$ (i.e., la complexité en nombre de mouvements du protocole $\mathcal{PIR}$).

Ensuite, au plus $S_1(\mathcal{B})$ règles de $\mathcal{B}$ sont exécutées entre chaque règle de $Reset(\mathcal{B})$. D'où, le délai pour démarrer un calcul demandé de $\mathcal{SSBB}(\mathcal{B})$ est borné par le produit de ces deux complexités et le théorème est vérifié. □

Suivant le même argument que pour la complexité en terme de rondes, nous avons le corollaire suivant :

Corollaire 6.3.2 *À partir de n'importe quelle configuration, un calcul demandé (et complet) de $\mathcal{SSBB}(\mathcal{B})$ est exécuté en $O(\Delta \times n^3 \times S_1(\mathcal{B}))$ mouvements.*

D'après le corollaire précédent, nous savons que tout calcul demandé de la tâche $\mathcal{TASK}$ est exécuté en un nombre fini de mouvements et que ce nombre est majoré par $\Delta \times n^3 \times S_1(\mathcal{B})$. Nous allons maintenant montrer que cette borne supérieure ne peut-être atteinte que lors de la première exécution demandée, les autres exécutions de la tâche $\mathcal{TASK}$ étant significativement meilleures.

Théorème 6.3.8 *Après que $\mathcal{SSBB}(\mathcal{B})$ est exécuté le premier calcul demandé, les autres calculs demandé sont exécutés en $O(S_2(\mathcal{B}) + n)$ mouvements.*

Preuve. Durant l'exécution du premier calcul demandé, le système effectue, à partir d'une configuration quelconque, une réinitialisation globales des variables de $\mathcal{B}$ puis une exécution de $\mathcal{B}$. Durant la réinitialisation des variables de $\mathcal{B}$, des règles de $\mathcal{B}$ peuvent être exécutées à cause de la configuration initiale quelconque. Suite à la réinitialisation, $\mathcal{B}$ s'exécute à partir d'une de ses configurations initiales normales. À partir d'une telle configuration, $\mathcal{B}.End(r)$ sera vérifié lorsque les variables de $\mathcal{B}$ auront atteintes une configuration terminale. Donc, l'exécution du prochain calcul demandé commencera à partir d'une configuration terminale de $\mathcal{B}$. La réinitialisation et l'exécution de $\mathcal{B}$ seront alors effectuées séquentiellement (r ne pouvant initier $\mathcal{B}$ avant la terminaison de la réinitialisation). La réinitialisation coûtera $O(n)$ mouvements d'après le théorème 6.2.2 et l'exécution de $\mathcal{B}$ coûtera $S_2(\mathcal{B})$ mouvements. □

Le théorème précédent montre que suite à l'exécution du premier calcul demandé, le temps d'exécution en nombre de mouvements des autres calculs demandés sera du même ordre que le temps d'exécution de $\mathcal{B}$ lorsqu'il démarre dans une configuration initiale normale. En effet, à partir d'une configuration initiale normale, le temps d'exécution de $\mathcal{B}$ est en $\Omega(n)$ mouvements. Cette observation n'est en revanche pas valable pour la complexité en nombre de rondes. Cependant, bien que la complexité en nombre de rondes de $Reset(\mathcal{B})$ soit dans le pire des cas en $O(n)$ rondes, en moyenne, elle passe à $\Theta(D)$ rondes où D est le diamètre du réseau (en moyenne, l'arbre contruit par la diffusion d'un PIR est presque un arbre couvrant en largeur). Or, à partir d'une configuration initiale normale, le temps d'exécution de $\mathcal{B}$ est en $\Omega(D)$ rondes. D'où le théorème suivant :

Théorème 6.3.9 *En moyenne, le temps d'exécution en nombre de rondes d'un calcul de $\mathcal{SSBB}(\mathcal{B})$ est du même ordre que le temps d'exécution de $\mathcal{B}$ lorsqu'il démarre dans une configuration initiale normale.*

6.4 Applications

Dans cette section, nous présentons deux protocoles. Ces deux protocoles ne sont ni auto-stabilisants ni instantanément stabilisants. Cependant, nous allons monter qu'ils vérifie la propriété *SSBB-Friendly* (définition 6.1.3, page 121). Donc, en composant ces protocoles avec notre transformateur ($\mathcal{SSBB}$), nous obtenons des protocoles instantanément stabilisants. Ces deux protocoles permettent de construire des protocoles de service instantanément stabilisants pour le parcours en profondeur et le calcul d'arbre couvrant en largeur.

Algorithm 6.4.17 Algorithme $\mathcal{DFS}0$ pour $p = r$

Entrée-sortie : $Request_p \in \{Wait,In,Out\}$ initialisé à Out ;

Entrée : $Neig_p$: ensemble des voisins (localement ordonné) ;

Constante : $Par_p = \perp$;

Variable : $S_p \in Neig_p \cup \{C,D\}$ initialisé à C ;

Macro :

$Next_p \quad = \quad (q = \min_{\prec_p}\{q' \in Neig_p :: S_{q'} = C\})$ **si** q **existe**, D **sinon** ;

Prédicats :

$Leaf(p) \quad \equiv \quad (\forall q \in Neig_p :: (Par_q = p) \Rightarrow (S_q = C))$
$Forward(p) \quad \equiv \quad (S_p = C)$
$Backward(p) \quad \equiv \quad (\exists q \in Neig_p :: S_p = q \wedge S_q = D)$
$Clean(p) \quad \equiv \quad (S_p = D) \wedge Leaf(p)$

Règles :

$F :: Forward(p) \wedge (Request_p = Wait) \rightarrow S_p := Next_p ; Request_p := In ;$
$B :: Backward(p) \rightarrow S_p := Next_p ;$
$C :: Clean(p) \rightarrow S_p := C ; Request_p := Out ;$

Algorithm 6.4.18 Algorithme $\mathcal{DFS}0$ pour $p \neq r$

Entrée : $Neig_p$: ensemble des voisins (localement ordonné) ;

Variables : $S_p \in Neig_p \cup \{C,D\}$ initialisé à C ; $Par_p \in Neig_p$ (non initialisé) ;

Macros :

$Pred_p \quad = \quad \{q \in Neig_p :: S_q = p\}$;
$Next_p \quad = \quad (q = \min_{\prec_p}\{q' \in Neig_p :: S_{q'} = C\})$ **si** q **existe**, D **sinon** ;

Prédicats :

$Leaf(p) \quad \equiv \quad (\forall q \in Neig_p :: (Par_q = p) \Rightarrow (S_q = C))$
$Forward(p) \quad \equiv \quad (|Pred_p| = 1) \wedge (S_p = C)$
$Backward(p) \quad \equiv \quad (\exists q \in Neig_p :: S_p = q \wedge S_q = D)$
$Clean(p) \quad \equiv \quad (S_p = D) \wedge Leaf(p) \wedge (\forall q \in Neig_p :: S_q \in \{C,D\})$

Règles :

$F :: Forward(p) \rightarrow S_p := Next_p ; Par_p := (q \in Pred_p) ;$
$B :: Backward(p) \rightarrow S_p := Next_p ;$
$C :: Clean(p) \rightarrow S_p := C ;$

6.4.1 Parcours en profondeur

Dans cette sous-section, nous prouvons que le protocole $\mathcal{DFS}3$ présenté dans les algorithmes 6.4.19 et 6.4.20 vérifie la propriété $SSBB\text{-}Friendly(\mathcal{DFST},DUF)$ où $\mathcal{DFST}$ (*i.e., depth-first search traversal*) correspond au parcours en profondeur. Ainsi, nous pourrons conclure que le protocole $\mathcal{SSBB}(\mathcal{DFS}3)$ est un protocole de parcours en profondeur instantanément stabilisant sous un démon distribué inéquitable. Le code de $\mathcal{DFS}3$ est très proche du code du protocole non-tolérant aux pannes $\mathcal{DFS}0$ présenté dans la section 5.3 (page 44). À titre de comparaison, nous rappelons le code de $\mathcal{DFS}0$ dans les algorithmes 6.4.17 et 6.4.18.

Tout d'abord, nous montrons qu'il existe une configuration à partir de laquelle $\mathcal{DFS}3$ exécute un parcours en profondeur dans le réseau. Considérons la configuration γ_α où $\forall p \in V$, $S_p = C$. Un exemple d'exécution de $\mathcal{DFS}3$ à partir de la configuration γ_α est montré dans la figure 6.4.2. À partir de γ_α, $\mathcal{DFS}3$ reprend les principes de la phase de visite du protocole non-tolérant aux pannes $\mathcal{DFS}0$. En fait, le protocole $\mathcal{DFS}3$ est plus simple que le protocole $\mathcal{DFS}0$: il ne comporte ni phase de nettoyage ni pointeur père. En effet, la phase de nettoyage et le pointeur père, qui est uniquement utilisé dans $\mathcal{DFS}0$ pour réaliser le nettoyage à partir des feuilles, sont inutiles ici car la réinitialisation des variables de $\mathcal{DFS}3$ est à la charge du protocole $Reset$ (à cet effet, nous avons défini la macro $Init_p$ pour chaque processeur p). Cependant, si l'on souhaite utiliser $\mathcal{SSBB}(\mathcal{DFS}3)$ pour construire un arbre couvrant en profondeur du réseau, un pointeur père peut-être affecté par chaque processeur

Algorithm 6.4.19 Algorithme $\mathcal{DFS}3$ pour $p = r$

Entrée-sortie : $Request_p \in \{Wait, In, Out\}$;

Entrée : $Neig_p$: ensemble des voisins (localement ordonné) ;

Variable : $S_p \in Neig_p \cup \{C, D\}$;

Macros :

$\mathit{Init}_p \quad = \quad S_p := C$;
$Pred_p \quad = \quad \{q \in Neig_p :: S_q = p\}$;
$Next_p \quad = \quad (q = \min_{\prec_p}\{q' \in Neig_p :: S_{q'} = C\})$ **si** q **existe**, D **sinon** ;

Prédicats :

$\mathit{End}(p) \quad \equiv \quad (S_p = D)$
$Error(p) \quad \equiv \quad (S_p \neq D) \wedge [(|Pred_p| \neq 0) \vee ((S_p = C) \wedge (\exists q \in Neig_p :: S_q \neq C))]$
$Forward(p) \quad \equiv \quad (S_p = C)$
$Backward(p) \quad \equiv \quad (\exists q \in Neig_p :: S_p = q \wedge S_q = D)$

Règles :

$E \quad :: \quad Error(p) \quad \rightarrow \quad S_p := D$;
$F \quad :: \quad Forward(p) \quad \rightarrow \quad S_p := Next_p$;
$B \quad :: \quad Backward(p) \quad \rightarrow \quad S_p := Next_p$;

Algorithm 6.4.20 Algorithme $\mathcal{DFS}3$ pour $p \neq r$

Entrée : $Neig_p$: ensemble des voisins (localement ordonné) ;

Variable : $S_p \in Neig_p \cup \{C, D\}$;

Macros :

$\mathit{Init}_p \quad = \quad S_p := C$;
$Pred_p \quad = \quad \{q \in Neig_p :: S_q = p\}$;
$Next_p \quad = \quad (q = \min_{\prec_p}\{q' \in Neig_p :: S_{q'} = C\})$ **si** q **existe**, D **sinon** ;

Prédicats :

$Error(p) \quad \equiv \quad (S_p \neq D)$
$\wedge [(|Pred_p| > 1) \vee (S_p \neq C \wedge |Pred_p| = 0) \vee ((S_p = C) \wedge (\exists q \in Neig_p :: S_q = D))]$
$Forward(p) \quad \equiv \quad (|Pred_p| = 1) \wedge (S_p = C)$
$Backward(p) \quad \equiv \quad (\exists q \in Neig_p :: S_p = q \wedge S_q = D)$

Règles :

$E \quad :: \quad Error(p) \quad \rightarrow \quad S_p := D$;
$F \quad :: \quad Forward(p) \quad \rightarrow \quad S_p := Next_p$;
$B \quad :: \quad Backward(p) \quad \rightarrow \quad S_p := Next_p$;

lors de la première réception du jeton (règle F) sans modifier la validité du protocole.

Comme pour $\mathcal{DFS}0$, la circulation réalisée par $\mathcal{DFS}3$ à partir de γ_α termine en r par l'affectation $S_r := D$. Après cette dernière affectation, plus aucun processeur n'est activable. À partir de γ_α et jusqu'à ce que la circulation termine, exactement une règle est exécutée à chaque mouvement : F or B. Lorsqu'un processeur p exécute l'une de ces règles :

- Soit il affecte D à S_p et devient inactivable pour toujours.
- Soit il désigne un voisin q tel que ($q \neq r \wedge S_q = C$) avec S_p et n'exécute plus de règles jusqu'à ce que $S_q = D$.

Donc, à partir de γ_α et jusqu'à ce que le système atteigne une configuration terminale, $n - 1$ règles sont exécutées pour désigner des successeurs et n règles sont exécutées pour affecter D aux variables S quel que soit le démon que nous supposons. D'où :

Lemme 6.4.1 *À partir de γ_α, $\mathcal{DFS}3$ exécute un parcours en profondeur en exactement $2n - 1$ mouvements (resp. rondes) quel que soit le démon.*

D'après le lemme 6.4.1, nous savons que le protocole $\mathcal{DFS}3$ vérifie le point 2 de la définition de $SSBB\text{-}Friendly$ (définition 6.1.3, page 121) pour un démon distribué inéquitable. Nous prouvons maintenant que le prédicat $\mathcal{DFS}3.End(r)$ (défini dans l'algorithme 6.4.19) vérifie la propriété

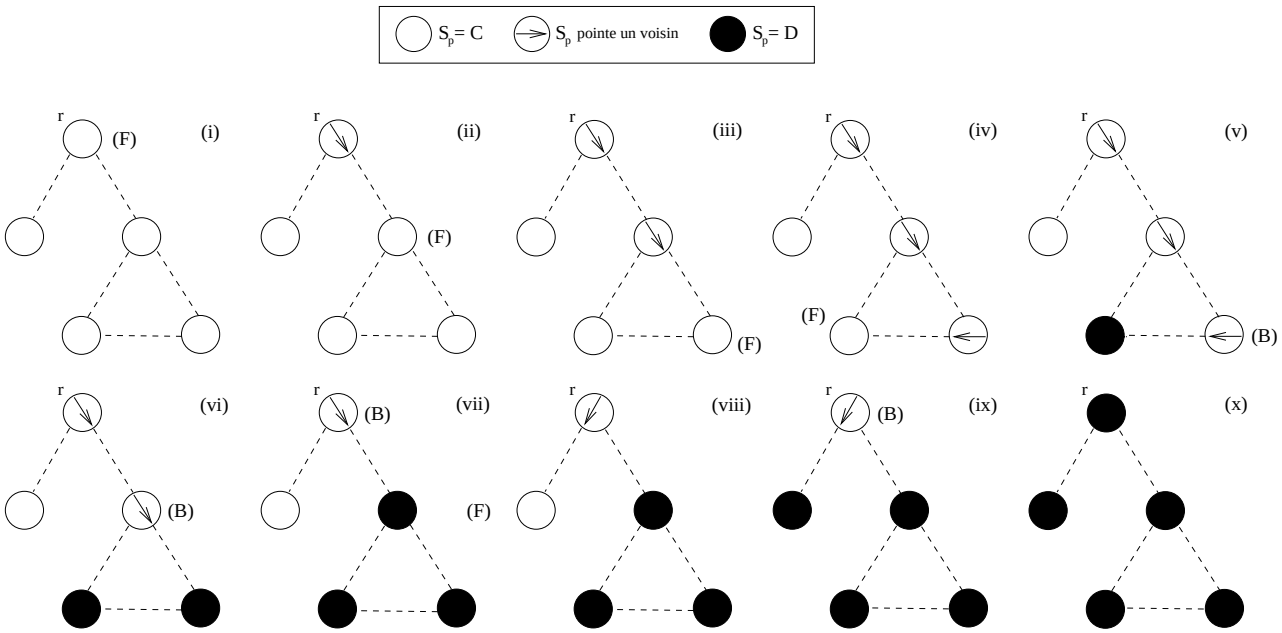


FIG. 6.4.2 – Exemple d'exécution de $\mathcal{DFS}3$ à partir de γ_α.

$BreakingIn(DUF)$ (point 1 de la définition de $SSBB\text{-}Friendly$, page 121). Pour montrer que le prédicat $\mathcal{DFS}3.End(r)$ vérifie $BreakingIn(DUF)$, il faut prouver que quelle que soit la configuration initiale (en particulier, quand la configuration initiale est différente de γ_α), $\mathcal{DFS}3$ converge en un nombre fini de mouvements vers une configuration terminale où $\mathcal{DFS}3.End(r)$ est vérifié (cf. conséquence 6.1.1). Le comportement de $\mathcal{DFS}3$ à partir d'une configuration différente de γ_α est le suivant : quand un processeur p détecte localement que le système n'est pas dans la configuration atteignable depuis γ_α, il affecte définitivement D à S_p par la règle E qui est la règle la plus prioritaire de $\mathcal{DFS}3$ (n.b. les priorités suivent l'ordre d'apparition des règles dans le texte des algorithmes). Deux exemples d'exécution de $\mathcal{DFS}3$ à partir d'une configuration différente de γ_α sont montrés dans les figures 6.4.3 et 6.4.4. Ci-dessous, nous allons montrer que, grâce à la règle E, et même si le système démarre d'une configuration différente de γ_α, le système atteint une configuration terminale où $\mathcal{DFS}3.End(r)$ est vérifié en un nombre fini de mouvements. Pour cela, nous allons utiliser le schéma de preuve suivant :

(i) Nous prouvons que toute exécution de $\mathcal{DFS}3$ contient un nombre fini de mouvements.
(ii) Puis, nous montrons que $\mathcal{DFS}3.End(r)$ est vérifié dans toute configuration terminale.

Lemme 6.4.2 *À partir de n'importe quelle configuration initiale, $\mathcal{DFS}3$ atteint une configuration terminale en au plus $(\Delta + 1) \times n$ mouvements.*

Preuve. Dans le protocole $\mathcal{DFS}3$, chaque fois qu'un processeur p exécute une règle soit il désigne un nouveau processeur avec S_p soit il affecte D à S_p.

- Si p affecte D à S_p, alors p devient inactivable pour toujours. Donc, $\forall p \in V$, p exécute $S_p := D$ au plus une fois durant l'exécution.
- p exécute $S_p := q$ avec $q \in Neig_p$ (via les règles F ou B) seulement si $S_q = C$. Ensuite, p sera à nouveau activable pour choisir un autre successeur (via B) seulement quand S_q sera égal à D. Or, lorsque $S_q = D$, q est inactivable pour toujours, donc, $S_q \neq C$ pour toujours et q ne peut plus jamais être désigné comme successeur par p. Ainsi, p peut successivement désigner chacun de ses voisins au plus une fois (*i.e.*, Δ règles) avant d'affecter D à S_p définitivement.

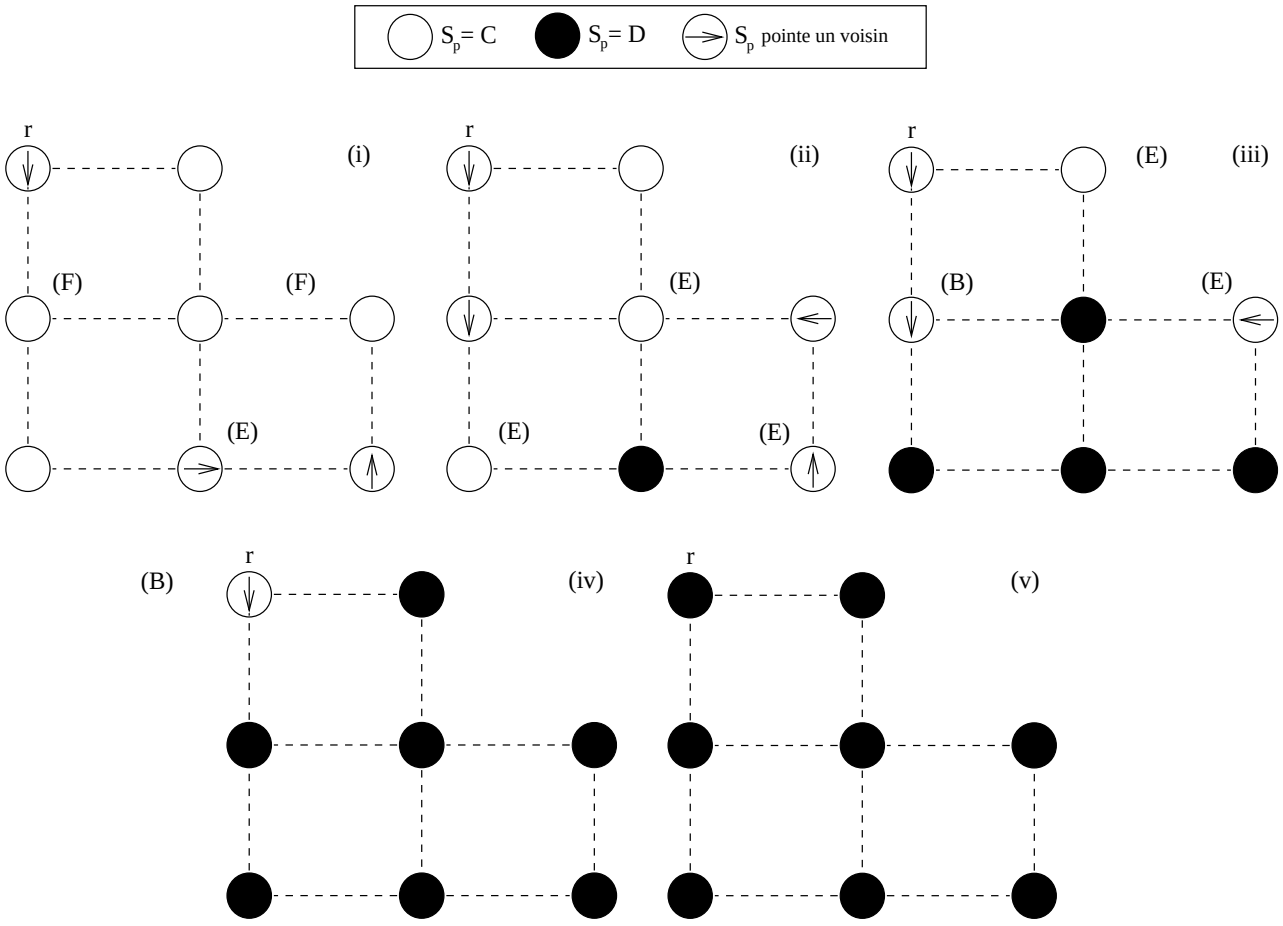


FIG. 6.4.3 – Exemple d'exécution de $\mathcal{DFS}3$ à partir d'une configuration différente de γ_α.

D'où, chaque processeur (n) exécute au plus $\Delta+1$ règles durant une exécution de $\mathcal{DFS}3$ et le lemme est vérifié. □

La notion de *chemin de successeurs* définie ci-dessous est utilisée dans le lemme suivant.

Définition 6.4.1 *[Chemin de Successeurs] Un chemin p_0, p_1, ..., p_k est appelé un* chemin de successeurs *si et seulement si* $\forall i$, $0 \leq i < k$, $S_{p_i} = p_{i+1}$.

Lemme 6.4.3 *Dans toute configuration terminale de* $\mathcal{DFS}3$, r *vérifie* $\mathcal{DFS}3.End(r)$.

Preuve. Supposons, par contradiction, qu'il existe une configuration γ où $\mathcal{DFS}3.End(r)$ n'est pas vérifié. Par contradiction, $S_r \neq D$ dans γ (cf. définition de $\mathcal{DFS}3.End(r)$ dans l'algorithme 6.4.19). Par contradiction encore, $\forall p \in V$, p vérifie $\neg Error(p)$ (sinon la règle E de p est activable). En particulier, cela signifie que $|Pred_r| = 0$ et $\forall p \in V \setminus \{r\}$, $(S_p \neq D) \Rightarrow (|Pred_p| \leq 1)$. Etudions maintenant les deux cas suivants :

1. $S_r = C$ dans γ. Comme *r* vérifie $\neg Error(r)$, nous savons que $\forall p \in Neig_r$, $S_p = C$. Dans ce cas, la règle F de *r* est activable et γ n'est pas une configuration terminale, contradiction.
2. $S_r = p$ tel que $p \in Neig_r$ dans γ. Nous savons que $|Pred_r| = 0$. De plus, tout processeur $p \neq r$ relié à *r* par un chemin de successeurs vérifie $(S_p \neq D) \Rightarrow (|Pred_p| = 1)$. Donc, nous pouvons déduire qu'il existe un chemin de successeurs ayant *r* comme extrémité initiale et dont l'extrémité finale $q \neq r$ vérifie $S_q \in \{C,D\}$:

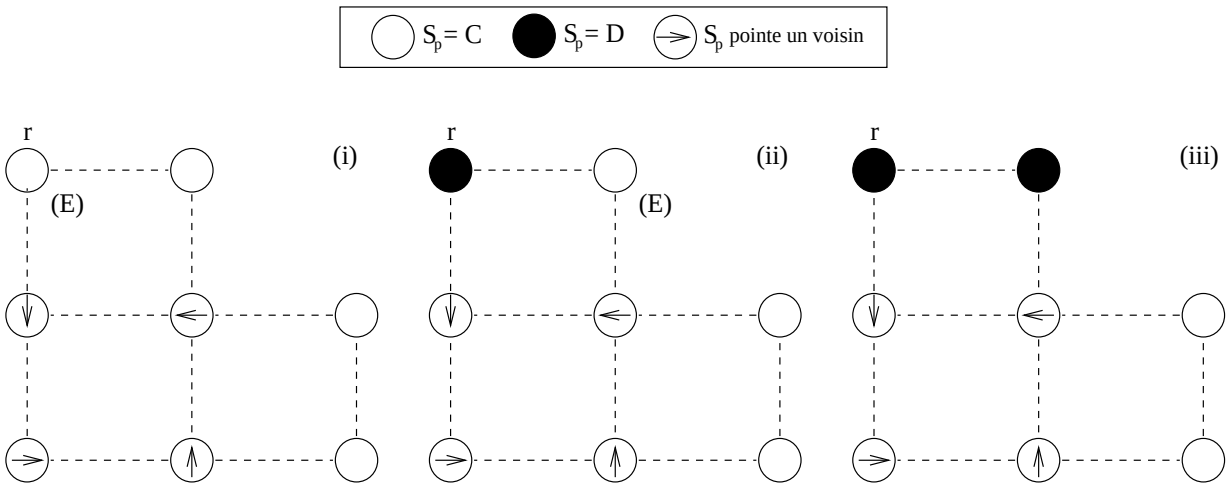


FIG. 6.4.4 – Deuxième exemple d'exécution de $\mathcal{DFS}3$ à partir d'une configuration différente de γ_α.

(a) Supposons que $S_q = C$. Comme, $|Pred_q| = 1$, la règle F de q est activable et γ n'est pas une configuration terminale, contradiction.
(b) Supposons que $S_q = D$. Dans ce cas, $\exists q' \in Neig_q$ tel que $S_{q'} = q$. Or, q' vérifie alors $Backward(q')$ (la garde de la règle B) et γ n'est pas une configuration terminale, contradiction.

Dans tous les cas, si $S_r \neq D$, alors γ n'est pas une configuration terminale. D'où, dans toute configuration terminale, $S_r = D$, *i.e.*, *r* vérifie $\mathcal{DFS}3.End(r)$. □

D'après les lemmes 6.4.2 et 6.4.3, le prédicat $\mathcal{DFS}3.End(r)$ vérifie les points 1 et 4 de la définition de la propriété *BreakingIn* (définition 6.1.2, page 119) sous un démon distribué inéquitable. De plus, nous savons déjà qu'à partir de γ_α, $\mathcal{DFS}3.End(r)$ est vérifié seulement quand le parcours est terminé (point 2 de la définition de *BreakingIn*). Finalement, d'après les règles des algorithmes 6.4.19 et 6.4.20, nous pouvons déduire que $\mathcal{DFS}3.End(r)$ vérifie le point 3 de la définition de *BreakingIn*. D'où, le lemme suivant :

Lemme 6.4.4 $\mathcal{DFS}3.End(r)$ *vérifie* $BreakingIn(DUF)$.

D'après les règles des algorithmes 6.4.19 et 6.4.20, nous pouvons déduire que $\mathcal{DFS}3$ vérifie le point 3 de la définition de la propriété $SSBB\text{-}Friendly$ (cf. définition 6.1.3, page 121). D'après les lemmes 6.4.1 et 6.4.4, donc :

Lemme 6.4.5 $\mathcal{DFS}3$ *vérifie* $SSBB\text{-}Friendly(DFST,DUF)$.

D'après le lemme 6.4.5 et le théorème 6.3.4, nous avons :

Théorème 6.4.1 $\mathcal{SSBB}(\mathcal{DFS}3)$ *est un protocole instantanément stabilisant de parcours en profondeur sous un démon distribué inéquitable.*

Nous pouvons remarquer que le code du protocole $\mathcal{DFS}3$ est très similaire au code de la solution non-tolérante aux fautes classique ($\mathcal{DFS}0$) et est plus simple que le code des protocoles des sections 5.4 et 5.5 ($\mathcal{DFS}1$ et $\mathcal{DFS}2$) ainsi que de n'importe quelle solution auto-stabilisante de la littérature. L'analyse de complexité de $\mathcal{SSBB}(\mathcal{DFS}3)$ fournie ci-dessous va révéler qu'en plus le protocole $\mathcal{SSBB}(\mathcal{DFS}3)$ résoud presque tous les inconvénients des solutions précédentes.

Analyse de complexité de SSBB(DFS3). Le lemme ci-dessous permet d'évaluer la complexité en nombre rondes du protocole $\mathcal{SSBB}(\mathcal{DFS}3)$.

Lemme 6.4.6 *À partir de n'importe quelle configuration initiale, $\mathcal{DFS}3$ atteint une configuration terminale en au plus $2n-1$ rondes.*

Preuve. D'après le lemme 5.5.18, nous savons qu'à partir de γ_α, le système atteint une configuration terminale en $2n-1$ rondes. Maintenant si la configuration initiale est différente de γ_α, les corrections effectuées par les règles E se font en parallèle et à chaque correction, le processeur corrigé ne peut plus jamais être visité. Les corrections permettant ainsi d'économiser des visites, le pire des cas pour atteindre une configuration terminale correspond en fait à une exécution commençant dans γ_α et le lemme est vérifié. □

Nous avons déjà proposé deux protocoles instantanément stabilisants de parcours en profondeur sous un démon distribué inéquitable pour des réseaux quelconques dans les sections 5.4 et 5.5. L'inconvénient de la première solution, *i.e.*, le protocole $\mathcal{DFS}1$, est son occupation mémoire $O(n \times \log n)$ bits par processeur. Dans la section 5.5, nous résolvons cet inconvénient en proposant le protocole $\mathcal{DFS}2$ qui lui utilise $O(\log n)$ bits par processeur. Ici, d'après le code des algorithmes 6.4.19 et 6.4.20, nous pouvons déduire que l'occupation mémoire de notre protocole $\mathcal{DFS}3$ est en $O(\log \Delta)$ bits par processeur. Donc, d'après le lemme 6.3.5, la solution instantanément stabilisante contruite avec notre transformateur, $\mathcal{SSBB}(\mathcal{DFS}3)$, est aussi efficace en mémoire que le protocole $\mathcal{DFS}2$, *i.e.*, $O(\log n)$ bits par processeur.

La complexité en temps du protocole $\mathcal{DFS}1$ présenté dans la section 5.4 est très bonne. En effet, en utilisant ce protocole, un parcours en profondeur complet demandé est exécuté en $O(n)$ rondes et $O(n^2)$ mouvements (le délai étant du même ordre). La complexité en temps du protocole $\mathcal{DFS}2$ est beaucoup moins bonne que la solution de la section 5.4 : un parcours en profondeur complet demandé est exécuté en $O(n^2)$ rondes et $O(\Delta \times n^3)$ mouvements (le délai est identique). Le délai du protocole $\mathcal{SSBB}(\mathcal{DFS}3)$ est en $O(n)$ rondes (d'après le théorème 6.3.6, les lemmes 6.4.1 et 6.4.6) et $O(\Delta^2 \times n^4)$ mouvements (d'après le théorème 6.3.7 et le lemme 6.4.2). Ensuite, à partir de n'importe quelle configuration, un parcours en profondeur complet demandé est exécuté par $\mathcal{SSBB}(\mathcal{DFS}3)$ en $O(n)$ rondes (d'après le corollaire 6.3.6, les lemmes 6.4.1 et 6.4.6) et $O(\Delta^2 \times n^4)$ mouvements (d'après le corollaire 6.3.7 et le lemme 6.4.2). La complexité en rondes de $\mathcal{SSBB}(\mathcal{DFS}3)$ est donc du même ordre que celle du protocole $\mathcal{DFS}1$. Cependant, il faut noter que la complexité en nombre de mouvements de $\mathcal{DFS}1$ reste meilleure (n.b. la complexité en nombre de mouvements de $\mathcal{DFS}2$ est aussi meilleure que celle de $\mathcal{SSBB}(\mathcal{DFS}3)$). En revanche, le coût en nombre de mouvements de $\mathcal{SSBB}(\mathcal{DFS}3)$ après le premier parcours devient du même ordre celui $\mathcal{DFS}1$: $O(n)$ mouvements (d'après le théorème 6.3.8 et le lemme 6.4.1). Ce coût étant d'ailleurs asymptotiquement optimal. Ainsi, $\mathcal{SSBB}(\mathcal{DFS}3)$ est un compromis quasi idéal entre les protocoles à listes ($\mathcal{DFS}1$) et à questions ($\mathcal{DFS}2$).

6.4.2 Construction d'arbre couvrant en largeur

Dans un réseau enraciné quelconque, un protocole de construction d'arbre couvrant en largeur calcule un arbre enraciné $Arbre(r)$ contenant tous les processeurs du réseau et tel que, $\forall p \in V$, la longueur du chemin élémentaire de p à r dans $Arbre(r)$ est égale à la distance entre p et r dans le réseau G (cf. chapitre A, page 159, pour les définitions). Dans la suite, nous noterons $BFSTC$ (*i.e., Breath-First Spanning Tree Construction*) la tâche consistant à construire un arbre couvrant en largeur du réseau enraciné en r.

Plusieurs protocoles auto-stabilisants de construction d'arbre en largeur ont été proposés dans la litérature : [AKY90, SS92, HC92, DIM93, Joh97]. Dans [AB98, DT01], les auteurs ont proposé un

Algorithm 6.4.21 Algorithme $\mathcal{BFS}$ pour $p = r$

Entrée-sortie : $Request_p \in \{Wait,In,Out\}$;

Entrée : $Neig_p$: ensemble des voisins (localement ordonné) ;

Constantes : $P_p = \perp$; $Level_p = 0$;

Variable : $Status_p \in \{C,R,F,D\}$;

Macros :

$\textbf{\textit{Init}}_p = Status_p := C$;

$Children_p = \{q \in Neig_p :: Status_q \neq C \wedge P_q = p\}$;

Prédicats :

$\textbf{\textit{End}}(p) \equiv (Status_p = D)$

$Error(p) \equiv (Status_p = C) \wedge (\exists q \in Neig_p :: Status_q \neq C)$

$Finish(p) \equiv (\forall q \in Children_p :: Status_q = D)$

$NewPhase(p) \equiv (Status_p = F) \wedge [\forall q \in Neig_p :: (Status_q \neq C) \wedge (q \in Children_p \Rightarrow Status_q \in \{B,D\})]$

$Start(p) \equiv (Status_p = C) \wedge (\forall q \in Neig_p :: Status_q = C)$

$ROK(p) \equiv (Status_p = R) \wedge (\forall q \in Children_p :: Status_q \in \{R,D\})$

$Forward(p) \equiv Start(p) \vee ROk(p)$

Règles :

Err	::	$Error(p)$	$\rightarrow$	$Status_p := D$;
New	::	$NewPhase(p) \wedge \neg Finish(p)$	$\rightarrow$	$Status_p := R$;
Fwd	::	$Forward(p)$	$\rightarrow$	$Status_p := F$;
D	::	$NewPhase(p) \wedge Finish(p)$	$\rightarrow$	$Status_p := D$;

schéma d'algorithme général permettant, entre autres, d'implémenter un protocole auto-stabilisant de construction d'arbre en largeur. Les protocoles proposés dans [SS92, AB98, Joh97, DT01] fonctionnent avec un démon distribué inéquitable. Cependant, excepté [Joh97], tous ces articles présentent des protocoles silencieux (cf. page 21). L'inconvénient de telles solutions est qu'aucun processeur n'est capable de détecter la terminaison du protocole sans utiliser de mécanisme externe (par exemple, utiliser un protocole de détection de terminaison). Le protocole proposé dans [Joh97] est un protocole à vague auto-stabilisant. Dans ce protocole, chaque vague termine à la racine. Cependant, comme ce protocole n'est pas instantanément stabilisant, la racine ne peut pas détecter quand le système vérifie ses spécifications. Donc, lorsqu'une vague termine à la racine, la racine ne peut pas détecter si un arbre couvrant en largeur du réseau a été effectivement calculé durant la vague ou pas. Finalement, il faut noter que le "transformateur" proposé dans [CDPV03] permet de construire un protocole à vague instantanément stabilisant de construction d'arbre en largeur à partir d'un protocole à vague construit pour un environnement sans fautes (par exemple [AG85]). Un avantage important d'une telle solution est que la racine est capable de détecter lorsqu'un arbre couvrant en largeur est disponible dans le réseau : dès la terminaison de la première vague initiée. Cependant, avec ce transformateur, les solutions obtenues ne fonctionnent qu'au plus avec un démon faiblement équitable et peuvent être très coûteuses en temps d'exécution.

Nous proposons maintenant d'écrire un protocole à vague instantanément stabilisant de construction d'arbre couvrant en largeur fonctionnant avec un démon distribué inéquitable. À cet effet, nous proposons un protocole appelé $\mathcal{BFS}$ et vérifiant $SSBB\text{-}Friendly(BFSTC,DUF)$. Le code de ce protocole (cf. algorithmes 6.4.21 et 6.4.22) est très proche de la solution non-tolérante aux pannes proposée par Awerbuch et Gallager dans [AG85]. Seuls les prédicats $Error(p)$ peuvent poser problème. Ils sont en fait très facile à déduire de l'algorithme de base.

Tout d'abord, nous montrons qu'il existe une configuration à partir de laquelle $\mathcal{BFS}$ construit un arbre couvrant en largeur du réseau. Considérons la configuration γ_α où, $\forall p \in V$, $Status_p = C$. Un exemple d'exécution de $\mathcal{BFS}$ à partir de la configuration γ_α est montré dans la figure 6.4.5. À partir de γ_α, $\mathcal{BFS}$ construit un arbre couvrant en largeur enraciné en r par phases : durant la $k^{\text{ème}}$ phase, tous les processeurs à distance k de r s'accrochent à l'arbre en construction. Puis, après l'accrochage de ces processeurs, r détecte en un temps fini la terminaison de la phase et initialise une nouvelle

Algorithm 6.4.22 Algorithme $\mathcal{BFS}$ pour $p \neq r$

Entrée : $Neig_p$: ensemble des voisins (localement ordonné) ;

Variables : $Status_p \in \{C,R,F,B,D\}$; $P_p \in Neig_p$; $Level_p \in \mathbb{N}$;

Macros :

Init$_p$	=	$Status_p := C$;
$Children_p$	=	$\{q \in Neig_p :: Status_q \neq C \wedge P_q = p\}$;
$Potential_p$	=	$\{q \in Neig_p :: Status_q = F\}$;

Prédicats :

$Finish(p)$	$\equiv$	$(\forall q \in Children_p :: Status_q = D)$
$Leaf(p)$	$\equiv$	$[\forall q \in Neig_p :: (P_q = p) \Rightarrow (Status_q \in \{C,D\})]$
$Hook(p)$	$\equiv$	$(Status_p = C) \wedge (\exists q \in Neig_p :: Status_q = F) \wedge Leaf(p)$
$NewPhase(p)$	$\equiv$	$(Status_p = B) \wedge (Status_{P_p} = R) \wedge (\forall q \in Children_p :: Status_q \in \{B,D\})$
$Forward(p)$	$\equiv$	$(Status_p = R) \wedge (Status_{P_p} = F) \wedge (\forall q \in Children_p :: Status_q \in \{R,D\})$
$Backtrack(p)$	$\equiv$	$(Status_p = F) \wedge [\forall q \in Neig_p :: (Status_q \neq C) \wedge (q \in Children_p \Rightarrow Status_q \in \{B,D\})]$
$GoodR(p)$	$\equiv$	$(Status_p = R) \Rightarrow (Status_{P_p} \in \{R,F\})$
$GoodF(p)$	$\equiv$	$(Status_p = F) \Rightarrow (Status_{P_p} = F)$
$GoodB(p)$	$\equiv$	$(Status_p = B) \Rightarrow (Status_{P_p} \notin \{C,D\})$
$GoodL(p)$	$\equiv$	$(Status_p \neq C) \Rightarrow (Level_p = Level_{P_p} + 1)$
$Bad(p)$	$\equiv$	$(\neg GoodR(p) \vee \neg GoodF(p) \vee \neg GoodB(p) \vee \neg GoodL(p))$
$Error(p)$	$\equiv$	$(Status_p \neq D) \wedge Bad(p)$

Règles :

Err	::	$Error(p)$	$\rightarrow$	$Status_p := D$;
Hk	::	$Hook(p)$	$\rightarrow$	$Status_p := B$; $P_p := \min_{\prec_p}(Potential_p)$; $Level_p := Level_{P_p} + 1$;
New	::	$NewPhase(p)$	$\rightarrow$	$Status_p := R$;
Fwd	::	$Forward(p)$	$\rightarrow$	$Status_p := F$;
Bck	::	$Backtrack(p) \wedge \neg Finish(p)$	$\rightarrow$	$Status_p := B$;
D	::	$Backtrack(p) \wedge Finish(p)$	$\rightarrow$	$Status_p := D$;

phase si la construction n'est pas terminée. Au début de la 1ère phase (*i.e.*, quand le système est dans γ_α), la règle Fwd de *r* est la seule règle activable du système. *r* initie donc le protocole en affectant F à $Status_r$ dans $\gamma_\alpha \mapsto \gamma_{\alpha+1}$. Ensuite, la règle Hk de chaque voisin de *r*, p, devient activable. En exécutant la règle Hk, chaque processeur p s'accroche à l'arbre de *r* : $Status_p$ reçoit B, P_p pointe *r* et $Level_p$ reçoit $Level_r + 1$, *i.e.*, 1. Quand tous les voisins de *r* ont rejoint l'arbre, *r* vérifie $NewPhase(r) \wedge \neg Finish(r)$, *i.e.*, la 1ère phase est terminée et *r* devient activable pour initier une nouvelle phase avec la règle New. *r* commence la 2ème phase par une réinitialisation des variables $Status$ de son arbre : R est affecté à $Status_r$. Ensuite, les voisins de *r* exécutent aussi leur règle New pour propager la réinitialisation. Quand tous les voisins de *r* ont affecté R à leur variable $Status$, *r* initie une nouvelle diffusion de la valeur F dans l'arbre (règle Fwd) pour signifier aux processeurs à distance 2 de la racine qu'ils doivent s'accrocher à l'arbre. Quand un voisin de *r*, p, reçoit une valeur F, deux cas sont alors possibles :

- p n'a aucun voisin à distance 2 de la racine. Dans ce cas, p vérifie $Backtrack(p) \wedge Finish(p)$ et affecte D à $Status_p$ via la règle D pour signifier que son sous-arbre est complètement calculé.
- p a au moins un voisin, q, à distance 2 de la racine. q rejoint alors l'arbre en s'accrochant à un de ses voisins appartenant à l'arbre et vérifiant $Status = F$ (règle Hk). Quand p détecte que tous ses voisins ont rejoint l'arbre, p affecte B à $Status_p$ par la règle Bck.

Quand tous les voisins de *r* ont affecté B ou D à leur variable $Status$, *r* détecte la fin de la 2ème phase et devient activable pour initier une nouvelle phase. Récursivement, les autres phases fonctionnent comme suit. *r* est activable pour initier la $k^{\text{ème}}$ phase (avec $k > 2$) quand $Status_r = F \wedge \forall p \in Neig_r$, $Status_p \in \{B,D\}$. La $k^{\text{ème}}$ phase commence par une réinitialisation initiée par *r* (règle New). Quand $\forall p \in Neig_r$, $Status_p \in \{R,D\}$, *r* initie une nouvelle diffusion de la valeur F dans l'arbre en construction pour signifier au processeur à distance k de la racine qu'ils doivent s'accrocher à l'arbre. La valeur est diffusée dans l'arbre comme suit : quand un processeur a son père dans l'arbre, P_q, tel

que $Status_{P_q} = F$, q attend que tous ses fils dans l'arbre (s'il y en a) vérifient $Status \in \{R,D\}$, et, ensuite, diffuse la valeur F à ses fils (cf. prédicat $Forward(q)$). Durant cette diffusion, un des deux cas suivants finit par apparaître dans chaque chemin de l'arbre :

- Une valeur F atteint un processeur p tel que tous ses fils dans l'arbre vérifient $Status = D$ (*i.e.*, p vérifie $Backtrack(p) \wedge Finish(p)$). p affecte alors D à $Status_p$ (règle D) pour signifier que son sous-arbre est complètement construit.
- Une valeur F atteint une feuille f de l'arbre. Si f n'a aucun voisin à distance k de la racine, f vérifie $Backtrack(p) \wedge Finish(p)$ et affecte D à $Status_f$ par la règle D. Sinon, f a au moins un voisin, q, tel que q est à distance k de *r*. Dans ce cas, chaque q rejoint l'arbre en s'accrochant à un processeur de l'arbre vérifiant $Status = F$ (règle Hk). Quand f détecte que tous ses voisins sont dans l'arbre (*i.e.*, f vérifie $Backtrack(f) \wedge \neg Finish(f)$), f affecte B à $Status_f$ par la règle Bck.

les valeurs B et D sont ensuite propagées dans l'arbre (via les règles Bck et D) et *r* finit par vérifier à nouveau $Status_r = F \wedge \forall p \in Neig_r$, $Status_p \in \{B,D\}$. Dans ce cas, si $\exists p \in Neig_r$ tel que $Status_p = B$, alors *r* initie une $(k+1)^{\text{ème}}$ phase (règle New). Sinon, l'arbre couvrant est terminé et *r* affecte D à $Status_r$ (règle D) pour signifier cette terminaison, *i.e.*, un arbre couvrant en largeur du réseau est maintenant disponible.

D'après la discussion précédente, à partir de γ_α, le protocole $\mathcal{BFS}$ construit un arbre couvrant en largeur par phases : durant la $k^{\text{ème}}$ phase, tous les processeurs à distance k de la racine s'accrochent à l'arbre. Maintenant, par définition, la hauteur d'un arbre couvrant en largeur est égale à D où D est le diamètre du réseau. Donc, à partir γ_α, $\mathcal{BFS}$ exécute exactement D phases pour construire l'arbre. De plus, nous pouvons remarquer que chaque phase est réalisée en $O(D)$ rondes et $O(n)$ mouvements. D'où, le lemme suivant :

Lemme 6.4.7 *À partir de γ_α, $\mathcal{BFS}$ construit un arbre couvrant en largeur en $O(D \times n)$ mouvements et $O(D^2)$ rondes quel que soit le démon.*

D'après le lemme 6.4.7, nous savons que le protocole $\mathcal{BFS}$ vérifie le point 2 de la définition $SSBB$-$Friendly$ (définition 6.1.3, page 121) avec un démon distribué inéquitable. Nous prouvons maintenant que le prédicat $\mathcal{BFS}.End(r)$ (défini dans l'algorithme 6.4.21) vérifie $BreakingIn(DUF)$ (point 1 de la définition $SSBB$-$Friendly$, page 121). Pour montrer que $\mathcal{BFS}.End(r)$ vérifie la propriété $BreakingIn(DUF)$, il faut prouver que quelle que soit la configuration initiale (en particulier, quand la configuration initiale est différente de γ_α), $\mathcal{BFS}$ converge en un nombre fini de mouvements vers une configuration terminale où $\mathcal{BFS}.End(r)$ est vérifié (cf. conséquence 6.1.1). Le comportement du protocole $\mathcal{BFS}$ à partir d'une configuration différente de γ_α est le suivant : quand un processeur p détecte localement que le système n'est pas dans une configuration atteignable depuis γ_α, il affecte définitivement D à $Status_p$ par la règle Err qui est la règle la plus prioritaire de $\mathcal{BFS}$. Ci-dessous, nous allons montrer que, grâce à la règle Err et même si le système démarre d'une configuration différente de γ_α, le système atteint une configuration terminale où $\mathcal{BFS}.End(r)$ est vérifié en un nombre fini de mouvements. Pour cela, nous allons utiliser le schéma de preuve suivant :

(i) Nous prouvons que toute exécution de $\mathcal{BFS}$ contient un nombre fini de mouvements.
(ii) Puis, nous montrons que $\mathcal{BFS}.End(r)$ est vérifié dans toute configuration terminale.

Pour cela, nous allons utiliser les définitions suivantes :

Définition 6.4.2 (CheminBFS(p)) *Pour tout processeur p tel que $Status_p \neq C$, $CheminBFS(p)$ est l'unique chemin $p_0, p_1, p_2, \ldots p_k = p$ vérifiant les conditions suivantes :*

1. $\forall i$, $1 \leq i \leq k$, $P_{p_i} = p_{i-1}$.

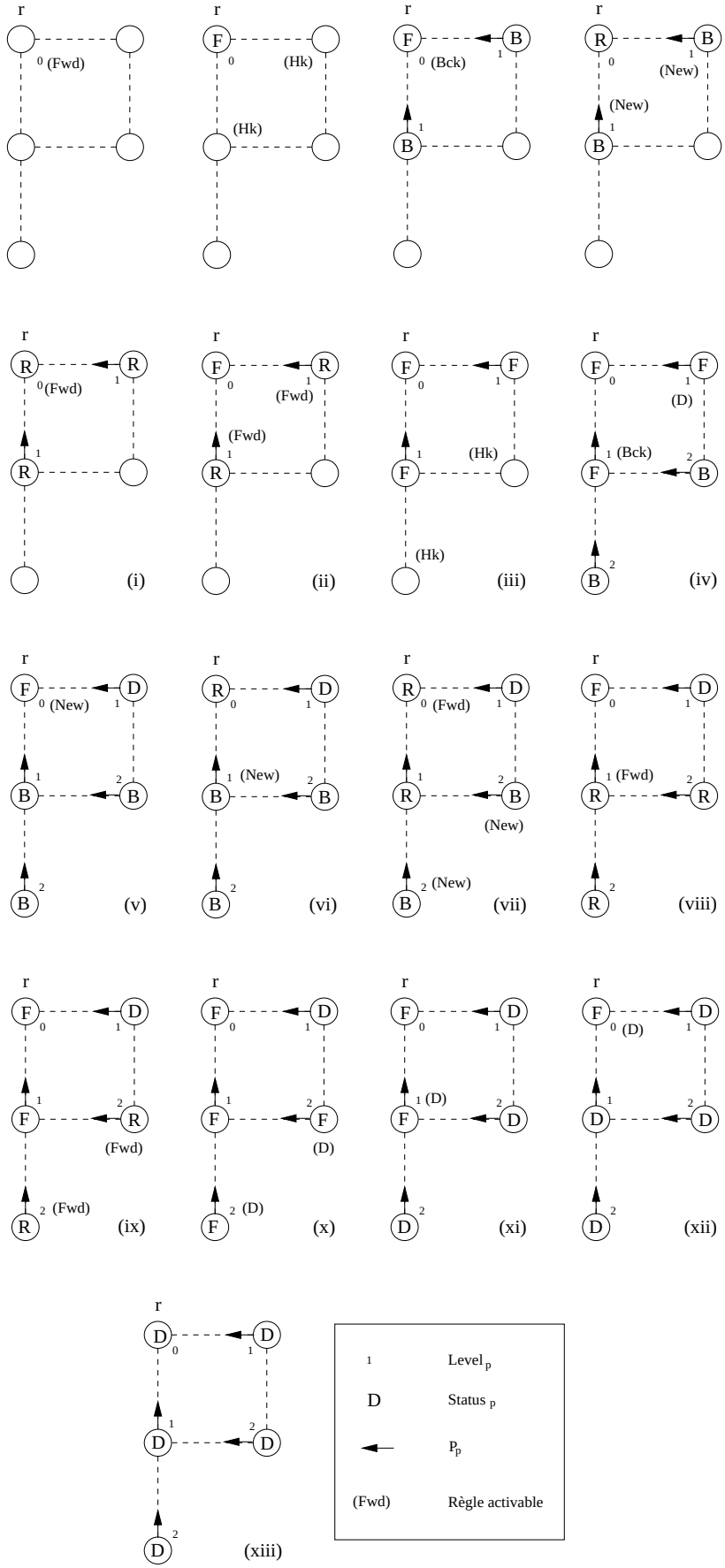


FIG. 6.4.5 – Exemple d'exécution de $\mathcal{BFS}$ à partir de γ_α.

2. $\forall i, 1 \leq i \leq k$, *(*$Status_{p_i} \neq C$*)* $\wedge \neg Bad(p_i)$.
3. *(*$p_0 \neq r$*)* $\Rightarrow Bad(p_0)$.

Définition 6.4.3 (ArbreBFS$_p$) *Pour tout processeur* p *tel que (*$p =$ r*)* $\vee$ $Bad(p)$*, nous définissons l'ensemble de processeurs* $ArbreBFS_p$ *de la manière suivante :* $\forall q \in V$, $q \in ArbreBFS_p$ *si et seulement si* $Status_q \neq C$ *et* p *est l'extrémité initiale de* $CheminBFS(q)$.

Remarque 6.4.1 *Dans toute configuration atteignable à partir de* γ_α*, nous avons* $\forall p \in V$*, (*$Status_p = C$*)* $\vee$ *(*$p \in ArbreBFS_r$*).*

Définition 6.4.4 (TraceStatus) *Soit* Y *un tuple de processeurs (*$Y = (p_0, p_1, \ldots, p_k)$*).* $TraceStatus(Y) = Status_0 Status_1 \ldots Status_k$ *est la liste des valeurs de la variable* $Status$ *des processeurs* p_i $(i = 0 \ldots k)$.

D'après la défintion 6.4.2, seule l'extrémité d'un $CheminBFS$ peit vérifié le prédicat Bad donc, nous déduisons le lemme suivant d'après la définition du prédicat Bad (cf. algorithme 6.4.22).

Lemme 6.4.8 $\forall p \in V$ *tel que* $Status_p \neq C$, $TraceStatus(CheminBFS(p)) \in F^+R^*B^*D^* \cup R^+B^*D^* \cup D^*$.

Lemme 6.4.9 *À partir de n'importe quelle configuration initiale,* $\mathcal{BFS}$ *atteint une configuration terminale en* $O(D \times n)$ *mouvements.*

Preuve. Soit $p \in V$ un processeur. Si $Status_p = C$, alors, après avoir exécuté une règle (Hk si $p \neq$ *r*, Fwd sinon), p vérifie $Status_p \neq C$ pour toujours. Ensuite, dès que $Status_p = D$, p n'est plus jamais activable. Donc, pour prouver ce lemme, nous allons montrer que chaque processeur p vérifiant $Status_p \notin \{C,D\}$ exécute $O(D)$ règles avant d'exécuter $Status_p := D$, *i.e.*, avant d'exécuter les règles D ou Err.

Soit $p \in V$ tel que $Status_p \in \{R,F,B\}$ et q l'extrémité initiale de $CheminBFS(p)$. Deux cas sont alors possibles :

a) $q \neq$ *r*. Dans ce cas, $p \neq$ *r* et nous étudions les trois possibilités suivantes :
   1. $Status_p = F$. Alors, d'après le lemme 6.4.8, $TraceStatus(CheminBFS(p)) = F^+$ et p ne peut pas exécuter de règle tant que $\neg Backtrack(p)$. Supposons donc que p finit par vérifier $Backtrack(p)$. p peut alors uniquement exécuter la règle Bck. En exécutant Bck, p affecte B à $Status_p$, la valeur B remonte alors dans $CheminBFS(p)$ et p n'est plus activable tant que $Status_{P_p} \neq R$. Or, $\forall p' \in CheminBFS(p)$, $p' \neq r \wedge Status_{p'} \neq R$ et *r* est le seul processeur capable de générer une valeur R. Donc, p ne peut plus exécuter de règle excepté les règles D ou Err. Ainsi, si $Status_p = F$, alors p peut au plus exécuter une règle avant d'exécuter la règle D ou la règle Err.
   2. $Status_p$=R. Alors, d'après le lemme 6.4.8, $TraceStatus(CheminBFS(p))$=$F^*R^+$.
      – Supposons que $Status_q = R$. Alors, $TraceStatus(CheminBFS(p)) = R^+$ et p n'est pas activable tant que $Status_{P_p} \neq F$. Or, $\forall p' \in CheminBFS(p)$, $p' \neq r \wedge Status_{p'} \neq F$ et *r* est le seul processeur capable de générer une valeur F. Donc, p n'exécute plus jamais de règle autre que les règles D ou Err.
      – Supposons que $Status_q = F$. Alors, p n'est pas activable tant que $Status_{P_p} \neq F$. Supposons que $Status_{P_p}$ finisse par être égale à F. Alors, la règle Fwd est la seule règle que p peut exécuter. Si p exécute la règle Fwd, alors $TraceStatus(CheminBFS(p))$ devient égal à F^+ et nous retrouvons le cas 1.

      Ainsi, si $Status_p = R$, alors p peut exécuter au plus deux règles avant d'exécuter la règle D ou Err.

3. $Status_p = B$. Alors, $TraceStatus(CheminBFS(p)) \in F^+R^*B^+ \cup R^+B^+$ d'après le lemme 6.4.8.
   - Supposons que $TraceStatus(CheminBFS(p)) = F^+R^*B^+$. Si $TraceStatus(CheminBFS(p)) = F^+R^+B^+$, alors les valeurs F et R descendent dans $CheminBFS(p)$ par l'exécution de règles New et Fwd jusqu'à ce que $Status_p = F$. Après que p ait exécuté ces deux règles, $Status_p = F$ et nous retrouvons le cas 1. Si $TraceStatus(CheminBFS(p)) = F^+B^+$, alors p n'est plus activable tant que $Status_{P_p} \neq R$. Or, $\forall p' \in CheminBFS(p)$, $p' \neq r \wedge Status_{p'} \neq R$ et r est le seul processeur capable de générer une valeur R. Donc, p n'exécute plus de règle avant d'exécuter la règle D ou la règle Err.
   - Supposons que $TraceStatus(CheminBFS(p)) = R^+B^+$. Dans ce cas, la valeur R descend dans $CheminBFS(p)$ jusqu'à ce que $Status_p = R$. Donc, p exécute au plus une règle New avant de vérifier le cas 2.

   Ainsi, si $Status_p = B$ alors, p peut au plus exécuter trois règles avant d'exécuter la règle D ou la règle Err.

D'où, si $q \neq r$, alors p peut au plus exécuter trois règles avant d'exécuter la règle D ou la règle Err.

b) $q = r$. Dans ce cas, d'après la définition 6.4.3, $p \in ArbreBFS_r$. Or, nous savons que $ArbreBFS_r$ est contruit par phases.
   - Si $p = q = r$ (*i.e.*, $CheminBFS(p)$ contient un seul processeur), alors, dans le pire des cas, p participe à toutes les phases. Or, pour chaque phase r exécute une règle New et une règle Fwd. De plus le nombre de phases exécutées est borné par D. Donc, si $p = r$, alors p exécute $O(D)$ règles avant d'exécuter $Status_p := D$ par la règle D.
   - Sinon, $p \neq r$ et, dans le pire des cas, p participe à $D-k$ phases complètes où k est la distance de p à r dans $ArbreBFS_r$. Or, à chaque phase complète où p participe, p exécute une règle New, une règle Fwd et une règle Bck. Donc, si $p \neq r$, alors p exécute $O(D)$ règles avant d'exécuter $Status_p := D$ par la règle D.

   Ainsi, si $q = r$, alors p exécute $O(D)$ règles avant d'exécuter $Status_p := D$ par la règle D.

D'où, $\forall p \in V$ tel que $Status_p \in \{R,F,B\}$, p exécute $O(D)$ règles avant d'exécuter $Status_p := D$ (par la règle D ou Err) et le lemme est vérifié. □

Lemme 6.4.10 *Dans toute configuration terminale de $\mathcal{BFS}$,* r *vérifie $\mathcal{BFS}.End(r)$.*

Preuve. Supposons, par contradiction, qu'il existe une configuration γ terminale où $\mathcal{BFS}.End(r)$ n'est pas vérifié. Par contradiction, $Status_r \neq D$ dans γ (cf. définition de $\mathcal{BFS}.End(r)$ dans l'algorithme 6.4.21) et $\forall p \in V$, p vérifie $\neg Error(p)$ (sinon, la règle Err de p est activable). Nous étudions alors la configuration γ en fonction de ces deux cas :

1. $Status_r = C$ dans γ. Comme nous avons $\neg Error(r)$, r vérifie $Start(r)$ et la règle Fwd de r est activable. Donc, γ n'est pas une configuration terminale, contradiction.
2. $Status_r \neq C$ dans γ. Dans ce cas, d'après le lemme 6.4.8, $\forall p \in ArbreBFS_r$, $TraceStatus(CheminBFS(p)) \in F^+R^*B^*D^* \cup R^+B^*D^* \cup D^*$. Soit $p \in ArbreBFS_r$.
   a) Supposons que $p \neq r$, $Status_p = B$, et $Status_{P_p} = R$ dans γ. Alors, $\forall q \in Children_p$, $Status_q \in \{B,D\}$ car $q \neq r \wedge \neg Error(q)$ et la règle New de p est activable. Ainsi, γ n'est pas une configuration terminale, contradiction.
   b) Supposons que $p \neq r$, $Status_p = R$, et $Status_{P_p} = F$ dans γ. Alors, $\forall q \in Children_p$, $Status_q \in \{R,B,D\}$ car $q \neq r \wedge \neg Error(q)$.
      - Si $\exists q \in Children_p$, $Status_q = B$ alors, d'après a), la règle New de q est activable et γ n'est pas une configuration terminale, contradiction.

- Sinon, $\forall q \in Children_p$, $Status_q \in \{R,D\}$ et la règle Fwd de p est activable. D'où, γ n'est pas une configuration terminale, contradiction.

c) Supposons que $Status_p = F$ dans γ. Sans perte de généralité, nous supposons que p vérifie $Level_p = \max(\{Level_q :: q \in ArbreBFS_r \wedge Status_q = F\})$. Alors, $\forall q \in Children_p$, $Status_q \in \{R,B,D\}$ car $q \neq r \wedge \neg Error(q)$. Si $\exists q \in Children_p$, $Status_q = R$, alors, d'après b), q est activable dans γ et γ n'est pas une configuration terminale, contradiction. Donc, $\forall q \in Children_p$, $Status_q \in \{B,D\}$.

- Supposons que $\forall p' \in Neig_p$, $Status_{p'} \neq C$. Si $p = r$, alors la règle New ou la règle D de p est activable, contradiction. Sinon ($p \neq r$), la règle Bck ou la règle D de p est activable. Ainsi, γ n'est pas une configuration terminale, contradiction.
- Supposons que $\exists p' \in Neig_p$ tel que $Status_{p'} = C$. Comme, $\forall p'' \in Neig_{p'}$, ($p'' \neq r$) $\Rightarrow \neg Error(p'')$, p' vérifie $Leaf(p')$ et sa règle Hk est activable. Ainsi, γ n'est pas une configuration terminale, contradiction.

D'où, par contradiction, $a)$, $b)$ et $c)$ implique que $\forall p \in ArbreBFS_r$, $TraceStatus(CheminBFS(p)) \in R^+D^* \cup D^*$ dans γ. Or, comme nous avons supposé que $Status_r \notin \{C,D\}$, $\forall p \in ArbreBFS_r$, $TraceStatus(CheminBFS(p)) \in R^+D^*$ avec, en particulier, $Status_r = R$. Comme $\forall p \in Children_r$, $Status_p \in \{R,D\}$ (n.b. $\forall p \in Children_r$, $p \neq r \wedge \neg Error(p)$), r vérifie aussi $ROk(r)$ et la règle Fwd de r est activable dans γ. Ainsi, γ n'est pas une configuration terminale, contradiction.

Ainsi, dans tous les cas, si $Status_r \neq D$ *i.e.*, si $\mathcal{BFS}.End(r)$ n'est pas satisfait, alors γ n'est pas une configuration terminale et le lemme est vérifié. □

D'après les lemmes 6.4.9 et 6.4.10, le prédicat $\mathcal{BFS}.End(r)$ vérifie les points 1 et 4 de la définition de *BreakingIn* (définition 6.1.2, page 119) sous un démon distribué inéquitable. De plus, il est évident qu'à partir de γ_α, $\mathcal{BFS}.End(r)$ est vérifié seulement si un arbre couvrant en largeur est disponible dans le réseau (point 2 de la définition de *BreakingIn*, page 119). Finalement, d'après les règles des algorithmes 6.4.21 et 6.4.22, nous pouvons déduire que $\mathcal{BFS}.End(r)$ vérifie le point 3 de la définition de *BreakingIn* (définition 6.1.2, page 119). D'où, le lemme suivant :

Lemme 6.4.11 *$\mathcal{BFS}.End(r)$ vérifie $BreakingIn(DUF)$.*

D'après les règles des algorithmes 6.4.21 et 6.4.22, nous pouvons déduire que $\mathcal{BFS}$ vérifie le point 3 de la définition de la propriété $SSBB\text{-}Friendly$ (définition 6.1.3, page 121). D'où, d'après les lemmes 6.4.7 et 6.4.11, nous avons :

Lemme 6.4.12 *$\mathcal{BFS}$ vérifie $SSBB\text{-}Friendly(BFSTC,DUF)$.*

D'après le lemme 6.4.12 et le théorème 6.3.4, nous avons :

Théorème 6.4.2 *$\mathcal{SSBB}(\mathcal{BFS})$ est un protocole à vague de construction d'arbre couvrant en largeur instantanément stabilisant sous un démon distribué inéquitable.*

Nous pouvons remarquer que le code du protocole $\mathcal{BFS}$ est très similaire au code de la solution non-tolérante aux fautes de Awerbuch et Gallagher ([AG85]) et est plus simple que le code du protocole à vague auto-stabilisant de Johnen ([Joh97]). L'analyse de complexité présentée ci-dessous va montrer qu'en plus $\mathcal{SSBB}(\mathcal{BFS})$ est efficace.

Analyse de complexité de SSBB(BFS). Nous commençons l'analyse de complexité avec le lemme suivant. Il permet d'évaluer sera utilisé la complexité du protocole $\mathcal{SSBB}(\mathcal{BFS})$ en nombre de rondes.

Lemme 6.4.13 *À partir de n'importe quelle configuration initiale, $\mathcal{BFS}$ atteint une configuration terminale en $O(D^2)$ rondes.*

Preuve. D'après le lemme 6.4.7, nous savons qu'à partir de γ_α, le système atteint une configuration terminale en $O(D^2)$ rondes. Maintenant si la configuration initiale est différente de γ_α, les corrections effectuées par les règles Err se font en parallèle et à chaque correction, le processeur corrigé ne peut plus jamais participer à aucune phase. Les corrections permettant ainsi d'écourter les phases, le pire des cas pour atteindre une configuration terminale correspond en fait à une exécution commençant dans γ_α. D'ù, le lemme est vérifié. □

D'après le code des algorithmes 6.4.21 et 6.4.22, nous pouvons déduire que l'occupation mémoire de $\mathcal{BFS}$ est $O(\log n)$ bits par processeur. Donc, d'après le lemme 6.3.5, $\mathcal{SSBB}(\mathcal{BFS})$ a besoin de $O(\log n)$ bits par processeur. Considérons maintenant la complexité en temps. Le délai pour démarrer un calcul d'arbre couvrant en largeur demandé, à partir de n'importe quelle configuration, est en $O(D^2 + n)$ rondes (d'après le théorème 6.3.6 et les lemmes 6.4.7 et 6.4.13) et $O(\Delta \times D \times n^4)$ mouvements (d'après le théorème 6.3.7 et le lemme 6.4.9). Finalement, à partir de n'importe quelle configuration, un calcul d'arbre couvrant en largeur complet est exécuté en $O(D^2+n)$ rondes (d'après le corollaire 6.3.6 et les lemmes 6.4.7 et 6.4.13) et $O(\Delta \times D \times n^4)$ mouvements (d'après le corollaire 6.3.7 et le lemme 6.4.9). Il faut cependant noter que le pire des cas de la complexité en nombre de mouvements de $\mathcal{SSBB}(\mathcal{BFS})$ correspond en fait à la première exécution, pour les autres exécutions, la complexité chute à $O(D \times n)$ mouvements (cf. théorème 6.3.8 et lemme 6.4.7). Enfin, en moyenne, le temps d'exécution de $\mathcal{SSBB}(\mathcal{BFS})$ est en $O(D^2)$ rondes (cf. théorème 6.3.9 et lemme 6.4.7), *i.e.*, la complexité de la solution non-tolérante aux pannes classique de Awerbuch et Gallager ([AG85]).

6.5 Extension : Exclusion Mutuelle

Dans la sous-section 6.4.1, nous avons vu que la stabilisation instantanée de $\mathcal{SSBB}(\mathcal{DFS}3)$ assure qu'à partir de n'importe quelle configuration et suite à une demande, un jeton est initié en un temps fini et ce jeton visite tous les processeurs du réseau en profondeur d'abord. Avec une légère modification, $\mathcal{SSBB}(\mathcal{DFS}3)$ peut être transformé en une circulation de jeton perpétuelle, *i.e.*, un protocole qui enchaîne séquentiellement et perpétuellement des circulations de jeton sans attendre que la racine ne reçoive de requête. Il suffit simplement de remplacer la règle externe IR par la règle suivante :

$$IR_{Cyclic} :: (\mathcal{DFS}3.Request_r = Out) \rightarrow \mathcal{DFS}3.Request_r := Wait;$$

Dans la suite de la section, nous noterons $\mathcal{SSBB}(Cyclic\text{-}\mathcal{DFS}3)$ la version perpétuelle du protocole $\mathcal{SSBB}(\mathcal{DFS}3)$.

L'*exclusion mutuelle* est une application classique des circulations de jetons perpétuelles. Dans le problème de l'*exclusion mutuelle*, nous supposons l'existence d'une portion de code particulière, appelée *section critique* (notée dans la suite $<SC>$). La section critique doit être exécutée par au plus un processeur à chaque étape de l'exécution. L'exclusion mutuelle peut être spécifiée comme suit : tout processeur qui demande la section critique entre en section critique en un temps fini (*vivacité*), et si un processeur demandeur est en section critique alors il est le seul processeur en section critique (*sûreté*). Nous pouvons remarquer que, dans un environnement sans fautes, cette propriété de sûreté est équivalente à la propriété suivante "jamais plus d'un processeur n'est en section critique" dans le sens où tout protocole qui vérifie la première propriété vérifie aussi la seconde et réciproquement

[Vil02] (en effet, dans un environnement sans faute seuls les processeurs demandeurs accèdent à la section critique). Évidemment, ce n'est pas le cas dans un environnement où des fautes peuvent arriver.

En utilisant une circulation de jeton perpétuelle, l'exclusion mutuelle peut être résolue comme suit : le jeton est vu comme un privilège et un processeur peut exécuter la section critique seulement lorsqu'il détient le jeton. Malheureusement, $\mathcal{SSBB}(Cyclic\text{-}\mathcal{DFS}3)$ n'est pas un protocole instantanément stabilisant d'exclusion mutuelle. En effet, à partir de n'importe quelle configuration initiale, plusieurs jetons non initiés peuvent circuler simultanément dans le réseau. Cependant, nous savons qu'à partir de n'importe quelle configuration initiale, chaque cycle de calcul de $\mathcal{SSBB}(Cyclic\text{-}\mathcal{DFS}3)$ initié par *r* est composé d'une réinitialisation instantanément stabilisante des variables de $\mathcal{DFS}3$ suivi par une circulation d'un jeton unique dans le réseau. Donc, après la première réinitialisation du réseau, la spécification de l'exclusion mutuelle est vérifiée pour toujours. Ainsi, $\mathcal{SSBB}(Cyclic\text{-}\mathcal{DFS}3)$ est un protocole auto-stabilisant d'exclusion mutuelle. De plus, la racine peut détecter quand le jeton est unique dans le réseau : dès que *r* a initié le protocole (donc provoquer une réinitialisation globale), le jeton qu'elle créera sera unique dans le réseau. Le théorème suivant (théorème 6.5.1) montre qu'il existe un moyen non seulement pour la racine mais aussi pour tous les autres processeurs de détecter quand le jeton qu'ils détiennent est unique. En effet, le théorème 6.5.1 montre que, quel que soit le processeur p, p n'a juste qu'à compter les jetons différents qu'il reçoit et, le troisième jeton reçu est unique dans le réseau.

Théorème 6.5.1 *$\forall p \in V$, à partir de n'importe quelle configuration initiale, p est sûr de recevoir un jeton qui est unique dans le réseau dès qu'il exécute sa règle F pour la troisième fois.*

Preuve. Pour exécuter sa règle F, un processeur p doit, en particulier, vérifier $S_p = C$ dans $\mathcal{DFS}3$ ($Forward(p)$) et $PIF_p = C$ dans $Reset(\mathcal{DFS}3)$ (à cause de $Ok(p)$ dans la composition $\mathcal{DFS}3 \oplus_{|Ok(p)} Reset(\mathcal{DFS}3)$). Après avoir exécuté sa règle F, p vérifie $S_p \neq C \wedge PIF_p = C$. D'après les règles de $Reset(\mathcal{DFS}3)$ et de $\mathcal{DFS}3$, nous pouvons alors remarquer que p vérifiera à nouveau $S_p = C$ (*i.e.*, p sera à nouveau prêt de recevoir un jeton) seulement après avoir exécuté la règle Fck de $Reset(\mathcal{DFS}3)$. Donc, p exécute au moins une règle Fck entre chaque exécution de la règle F et, à partir du $3^{\text{ème}}$ jeton reçu, au moins deux règles Fck ont déjà été exécutées par p. Or, dès la $2^{\text{ème}}$ exécution de la règle Fck après la première exécution de la règle F, les règles Fck exécutées par p correspondent toujours à une phase de retour associé à une diffusion initiée par *r* d'après le point 4 de la propriété 6.2.1. Durant une telle phase de retour, chaque processeur q réinitialise S_q avec C (cf. $\mathcal{DFS}3.Init_q$) et affecte F à la variable PIF_q de $Reset(\mathcal{DFS}3)$. Ensuite, comme tout processeur q ne peut pas modifier S_q tant que $PIF_q \neq C$, le système fini par atteindre une configuration où $\forall q \in V$, $S_q = C$ d'après le point 1 de la propriété 6.2.1. Or, dans une telle configuration, toutes les variables de $\mathcal{DFS}3$ sont initialisées. D'où, à partir d'une telle configuration, au plus un jeton existe dans le réseau et le théorème est vérifié. □

D'après le théorème 6.5.1, nous pouvons trivialement obtenir un protocole auto-stabilisant d'exclusion mutuelle très efficace : il suffit d'autoriser les processeurs demandeurs à exécuter la section critique seulement lorsqu'ils recoivent un nouveau jeton (règle F) et, dans ce cas, tout processeur exécute au plus deux fois la section critique sans vérifier la propriété de sûreté de l'exclusion mutuelle.

Nous décrivons maintenant comment construire un protocole instantanément stabilisant d'exclusion mutuelle en utilisant $Cyclic\text{-}SSBB(\mathcal{DFS}3)$. D'après le théorème 6.5.1, nous savons qu'à partir du moment où un processeur devient demandeur de la section critique, il suffit qu'il attend de recevoir au moins trois jetons et ensuite d'exécuter la section critique lorsqu'il reçoit le $3^{\text{ème}}$ jeton par la règle F pour vérifier la sûreté de l'exclusion mutuelle. C'est la solution que nous proposons maintenant. Afin de simplifier, l'écriture de notre solution, nous utilisons la *composition conditionnelle*

(cf. définition 5.7.1, page 100). Nous noterons qu'il s'agit là de la première utilisation d'un comptage explicite dans le cadre d'un protocole instantanément stabilisant. Cependant, n'oublions pas que pour tout protocole instantanément stabilisant, il existe un comptage implicite illustré par la propriété que tout protocole initialisé après la configuration initiale (autrement dit la première exécution réellement initiée par le système) fonctionne suivant la spécification requise. Cette utilisation de comptage ne fait appel à aucune modification du modèle contrairement à [Pil05] où l'existence d'une variable inviolable sur au moins un processeur est requise.

En utilisant la composition conditionnelle, notre protocole d'exclusion mutuelle instantanément stabilisant correspond au protocole composite suivant : $\mathcal{ME} \circ|_{\{F-Guard(p)\}} Cyclic\text{-}\mathcal{SSBB}(\mathcal{DFS}3)$ (cf. l'algorithme 6.5.23 pour la description formelle de $\mathcal{ME}$). Dans ce protocole, $F\text{-}Guard(p)$ est la garde de la règle F de p dans $Cyclic\text{-}\mathcal{SSBB}(\mathcal{DFS}3)$, *i.e.*, $F\text{-}Guard(p) \equiv (Forward(p) \wedge Ok(p))$. Ensuite, il est important de noter que, contrairement aux autres protocoles présentés dans cette thèse, $\mathcal{ME} \circ|_{\{F-Guard(p)\}} Cyclic\text{-}\mathcal{SSBB}(\mathcal{DFS}3)$ est multi-initiateur : en effet, dans le problème de l'exclusion mutuelle, tout processeur peut être demandeur de la section critique. Donc, dans la suite, nous allons utiliser la variable $\mathcal{ME}.Request_p \in \{Wait,In,Out\}$ pour gérer la demande de section critique et cette variable sera définie pour chaque processeur au lieu de *r* seulement.

Algorithm 6.5.23 Algorithme $\mathcal{ME}$, $\forall p \in V$:

Entrée-sortie : $\mathcal{ME}.Request_p \in \{Wait,In,Out\}$;

Entrée : $F\text{-}Guard(p)$: garde de la règle F dans $\mathcal{SSBB}(\mathcal{DFS}3)$;

Variable : $Cnt_p \in \{1,2\}$;

Règles :

ME_1	::	$F\text{-}Guard(p) \wedge (\mathcal{ME}.Request_p = Wait)$	$\rightarrow$	$Cnt_p := 1$; $\mathcal{ME}.Request_p := In$;
ME_2	::	$F\text{-}Guard(p) \wedge (\mathcal{ME}.Request_p = In) \wedge (Cnt_p = 1)$	$\rightarrow$	$Cnt_p := 2$;
ME_3	::	$F\text{-}Guard(p) \wedge (\mathcal{ME}.Request_p = In) \wedge (Cnt_p = 2)$	$\rightarrow$	$<SC>$; $\mathcal{ME}.Request_p := Out$;

En fait, $\mathcal{ME} \circ|_{\{F-Guard(p)\}} Cyclic\text{-}\mathcal{SSBB}(\mathcal{DFS}3)$ fonctionne comme suit :

- Quand un processeur p est en attente de la section critique (*i.e.*, $\mathcal{ME}.Request_p = Wait$), il attend de recevoir un nouveau jeton par l'exécution de la règle F. Lors de l'exécution de la première règle F, p exécute dans le même mouvement la règle ME_1. Par la règle ME_1, p initialise le compteur local Cnt_p à 1 (pour se souvenir qu'il a déjà reçu un jeton) et affecte In à $\mathcal{ME}.Request_p$ (pour signifier que la demande de section critique a été prise en compte).
- Quand p reçoit un autre jeton en exécutant la règle F, il vérifie aussi $(\mathcal{ME}.Request_p = In) \wedge (Cnt_p = 1)$. Il exécute alors la règle ME_2 dans le même mouvement que la règle F : par cette règle, il incrémente simplement le compteur local Cnt_p.
- Finallement, par la règle ME_3, p exécute la section critique ($<SC>$) et, ensuite, affecte Out à $\mathcal{ME}.Request_p$ (pour signifier que la section critique demandée par p a été exécutée) dans le même mouvement que la troisième règle F.

Par ce mécanisme et le théorème 6.5.1, il est clair qu'à partir de n'importe quelle configuration, la propriété de sûreté de l'exclusion mutuelle est toujours vérifié. Ensuite, la propriété de vivacité est trivialement assurée par le fait que le protocole $Cyclic\text{-}\mathcal{SSBB}(\mathcal{DFS}3)$ garantit que tout processeur reçoit des jetons infiniment souvent même si le démon est inéquitable. D'où, nous concluons avec le théorème suivant :

Théorème 6.5.2 $\mathcal{ME} \circ |_{\{F-Guard(p)\}} Cyclic\text{-}\mathcal{SSBB}(\mathcal{DFS}3)$ *est un protocole d'exclusion mutuelle instantanément stabilisant sous un démon inéquitable.*

Il est important de noter qu'à partir de n'importe quelle configuration initiale, notre protocole instantanément stabilisant d'exclusion mutuelle n'empêche pas plusieurs processeurs d'exécuter la section

critique simultanément. En revanche, notre protocole assure que tout processeur demandeur exécutera la section critique de manière exclusive. Donc, lorsque plusieurs processeurs exécutent la section critique simultanément, aucun d'entre eux n'a effectué de demande (*i.e.*, aucun d'entre eux n'a exécuté la règle IR). Ainsi, lorsqu'un processeur devient demandeur (lorsqu'il exécute la règle externe IR), il ne rentre en section critique que lorsque la section critique est libre et qu'aucun autre processeur ne peut avoir le privilège d'y accéder, *i.e.*, lorsque le demandeur détient un jeton qui est unique dans le réseau.

6.6 Conclusion

Dans ce chapitre, nous avons présenté une méthode semi-automatique permettant de transformer un protocole non-tolérant aux fautes pour réseau quelconque en protocole instantanément stabilisant. Le protocole initial (*i.e.*, le protocole à transformer) doit être un protocole de service mono-initiateur où les règles de décision sont exécutées par l'initiateur uniquement.

Cette méthode nous avait déjà permis (dans [CDV06a]) de rendre instantanément stabilisant les protocoles à vagues auto-stabilisants mono-initiateurs où les règles de décision sont exécutées par l'initiateur uniquement. En outre, nous avions montré qu'appliquer aux protocoles auto-stabilisants, cette méthode permettait d'obtenir des protocoles transformés plus efficaces que les protocoles initiaux : par exemple, nous avions montré qu'en transformant le protocole de circulation de jeton auto-stabilisant de Huang et Chen ([HC93]) nous obtenions un protocole instantanément stabilisant fonctionnant avec un démon plus général (le démon inéquitable à la place du démon faiblement équitable) et avec de meilleures complexités en temps (la première exécution correcte du protocole transformé est en $O(n)$ rondes alors que le temps de stabilisation du protocole initial est en $\Omega(D \times n)$ rondes).

Contrairement à la solution de Cournier *et al* ([CDPV03]), notre méthode n'utilise pas de calculs périodiques d'état global du système pour assurer la propriété de stabilisation instantanée. En revanche, elle nécessite de modifier légèrement le code du protocole initial afin d'obtenir un protocole vérifiant une propriété supplémentaire. Cependant, cette propriété est plus facile à obtenir que la stabilisation (auto-stabilisation ou stabilisation instantanée). L'avantage de notre méthode par rapport à celle de [CDPV03] est que nous pouvons borner le surcoût de notre transformateur. En conséquence et contrairement aux solutions précédentes, nous pouvons écrire des protocoles instantanément stabilisants fonctionnant avec un démon distribué inéquitable : le démon le plus faible du modèle à états.

Afin de montrer la validité de notre méthode, nous avons ensuite proposé deux applications construites avec notre transformateur : un protocole de parcours en profondeur instantanément stabilisant et un protocole instantanément stabilisant de calcul d'arbre couvrant en largeur avec détection de terminaison. Ces deux protocoles fonctionnent avec un démon distribué inéquitable et sont efficaces en complexité en temps et en espace. De plus, ces deux exemples montrent que la propriété demandée par notre transformateur ne mène pas à complexifier les codes des protocoles comme le demande les solutions stabilisantes (auto-stabilisantes ou instantanément stabilisantes). En fait, les codes des protocoles restent très proches des solutions basiques non-tolérantes aux fautes. Ce qui montre qu'en pratique aussi la propriété $SSBB\text{-}Friendly$ (condition nécessaire et suffisante pour transformer des protocoles avec le protocole $\mathcal{SSBB}$) est plus simple à obtenir que l'auto-stabilisation ou la stabilisation instantanée. Enfin, grâce à la propriété de comptage due à notre transformateur, nous avons montré comment utiliser notre protocole de parcours en profondeur instantanément stabilisant pour résoudre l'exclusion mutuelle de manière instantanément stabilisante. En fait, cette propriété permet à tout processeur de détecter quand il détient un jeton unique dans le réseau. Ainsi, en utilisant cette méthode de détection simple, nous obtenons trivialement un protocole d'exclusion

mutuelle instantanément stabilisant.

Chapitre 7

Conclusion

Dans cette thèse, nous nous sommes intéressés au concept de stabilisation instantanée. Ainsi, nous avons tout d'abord proposé deux solutions instantanément stabilisantes au problème classique du *parcours en profondeur* distribué pour des réseaux enracinés quelconques (chapitre 5). Les deux protocoles que nous avons proposés sont écrits dans le modèle à états et fonctionnent sous l'hypothèse d'un démon distribué inéquitable : le démon le plus général de ce modèle.

Le premier est basé sur des listes d'identités. Chaque processeur maintient une liste d'identités durant le parcours. Ces listes d'identités représentent une mémoire des processeurs visités par le parcours courant. Ce protocole n'utilise pas de structures sous-jacentes telles que, par exemple, un arbre couvrant et n'a pas besoin de connaissances globales sur le réseau telles que sa taille, par exemple. Cependant, l'inconvénient majeur de ce protocole est lié à l'utilisation des listes d'identités. En effet, à cause de ces listes, l'occupation mémoire de ce protocole est importante : $O(n \times \log n)$ bits par processeur. En revanche, le protocole a de nombreux avantages. En effet, l'analyse de complexité du protocole montre qu'il est très efficace en temps. Par exemple, il exécute un parcours complet (conforme aux spécifications) en $O(n)$ rondes et $O(n^2)$ mouvements, respectivement. De plus, après le premier parcours initié, les autres parcours sont effectués en $O(n)$ mouvements. Enfin, le surcoût en temps de ce protocole est négligeable : en effet, à partir d'une configuration initiale normale, son temps d'exécution est quasiment identique à celui du protocole non tolérant aux fautes le plus efficace ($\Theta(2(n-1))$ mouvements et rondes). Nous pouvons en outre remarquer que, dans ce protocole, tout parcours initié par la racine visite toujours les noeuds dans le même ordre : notre protocole réalise des parcours en profondeur *premiers* du réseau. En effet, grâce aux liste d'identités, un processeur p, lorsqu'il détient le jeton initié par la racine, désigne toujours comme successeur son voisin non-visité minimal dans l'ordre local $\prec_p$ (si un tel processeur existe). D'où, l'arbre couvrant construit lors de la première exécution de $\mathcal{DFS}1$ est un point fixe. Cette solution a donné lieu à deux publications : [CDPV04, CDPV06].

Le second protocole utilise un principe de question/réponse pour remplacer les listes d'identités afin d'obtenir une occupation mémoire plus faible. Comme pour la première solution proposée, ce protocole n'utilise pas de structures sous-jacentes et n'a pas besoin de connaissances globales sur le réseau. Cependant, ce protocole est moins efficace que le précédent en temps d'exécution : par exemple, le délai est en $O(n^2)$ rondes et $O(\Delta \times N^3)$ mouvements alors que nous obtenons pour la même tâche $O(n)$ rondes et $O(n^2)$ mouvements avec la première solution. Le temps d'exécution des parcours et le surcoût en temps sont aussi moins efficaces. En revanche, cette nouvelle solution apporte des améliorations majeures. Premièrement, elle n'utilise plus les identités des processeurs. De ce fait, son occupation mémoire est fortement améliorée : $O(\log n)$ bits par processeur au lieu de $O(n \times \log n)$ pour la solution précédente. Ensuite, le protocole détecte efficacement les parcours corrompus : un processeur peut participer au plus deux fois à un parcours corrompu sans le détecter. Cette deuxième solution a été présentée lors de la conférence internationale *SSS'05* ([CDV05b]).

Nous avons ensuite proposé deux applications instantanément stabilisantes pouvant être obtenues à partir de nos deux protocoles de parcours en profondeur. Ces deux applications permettent de calculer des propriétés globales sur le réseau. La première application (cf. section 5.7.3) est un calcul de point fixe avec détection de termination. L'algorithme présenté permet de marquer les *points d'articulation* et les *isthmes* du réseau. La seconde application (cf. section 5.7.4) permet d'évaluer si un ensemble fourni par une application est un *ensemble séparateur* du réseau. Cette seconde application a été présentée lors de la conférence internationale $HiPC'05$ [CDV05a]. Les propriétés du protocole de parcours instantanément stabilisant permettent de calculer ses applications avec une détection de terminaison à la racine : à partir de n'importe quelle configuration initiale, suite à une demande à la racine, le calcul demandé est initié en un temps fini et lorsque le calcul termine, la racine décide correctement. Contrairement à une solution auto-stabilisante, une seule exécution du protocole est donc nécessaire pour que la racine décide correctement. En outre, il faut noter que les propriétés que nous utilisons dans ces protocoles peuvent être facilement adaptées pour des procotoles auto-stabilisants. Notamment, nous avons présenté dans [Dev05] un protocole silencieux auto-stabilisant d'identification des *points d'articulation* et des *isthmes* du réseau utilisant les mêmes propriétés que la première application. Ces applications ont un surcoût en temps négligeable par rapport au protocole de parcours en profondeur utilisé. Ensuite, elles ne requièrent que $O(\log n)$ bits supplémentaires par processeur. Notamment, la complexité en espace de notre protocole d'identification des *points d'articulation* et des *isthmes* du réseau en fait le protocole le plus efficace de la littérature. Enfin, nous pouvons remarquer que grâce aux propriétés de nos protocoles de parcours en profondeur instantanément stabilisants, ces applications sont aussi simples à réaliser que des solutions non-tolérantes aux fautes. En effet, les comportements anormaux dûs aux configurations initiales quelconques sont gérés par le protocole de parcours en profondeur instantanément stabilisant.

Dans la dernière partie de nos résultats (chapitre 6), nous adoptons une approche plus générale en présentant une méthode semi-automatique permettant de transformer un protocole non-tolérant aux fautes pour réseau quelconque en protocole instantanément stabilisant. Ce transformateur est fondé sur le protocole de PIR instantanément stabilisant que nous avons présenté lors de la conférence internationale *ICPADS'06* [CDV06b]. Dans notre transformateur, le protocole initial (*i.e.*, le protocole à transformer) doit être un protocole de service mono-initiateur où les règles de décision apparaissent uniquement dans le code de l'initiateur. Cette méthode nous avait déjà permis (dans [CDV06a]) de rendre instantanément stabilisant des protocoles auto-stabilisants tout en les améliorant : par exemple, nous avions montré qu'en transformant le protocole de circulation de jeton auto-stabilisant de Huang et Chen ([HC93]) nous obtenions un protocole instantanément stabilisant fonctionnant avec un démon plus général (le démon inéquitable à la place du démon faiblement équitable) et avec de meilleures complexités en temps (la première exécution correcte du protocole transformé est en $O(n)$ rondes alors que le temps de stabilisation du protocole initial est en $\Omega(D \times n)$ rondes). Contrairement aux solutions précédentes, notre méthode n'utilise pas de calculs périodiques d'état global du système pour assurer la propriété de stabilisation instantanée. En revanche, notre méthode nécessite de modifier légèrement le code du protocole initial afin d'obtenir un protocole vérifiant une propriété supplémentaire. Cependant, cette propriété est plus facile à obtenir que la stabilisation (auto-stabilisation ou stabilisation instantanée). L'avantage de notre méthode par rapport à celle de [CDPV03] est que nous pouvons maintenant borner le surcoût de notre transformateur. En conséquence et contrairement aux solutions précédentes, nous pouvons écrire des protocoles instantanément stabilisants fonctionnant avec un démon distribué inéquitable : le démon le plus général du modèle à états. Afin de montrer la validité de notre méthode, nous avons ensuite proposé deux applications construites avec notre transformateur : un protocole de parcours en profondeur instantanément stabilisant et un protocole instantanément stabilisant de calcul d'arbre couvrant en largeur avec détection de terminaison. Ces deux protocoles fonctionnent avec un démon distribué inéquitable et sont efficaces à la fois en complexité en temps et en espace. De plus, ces deux exemples montrent

que la propriété demandée par notre transformateur ne mène pas à complexifier les codes des protocoles comme le demande les solutions stabilisantes (auto-stabilisantes ou instantanément stabilisantes). En fait, les codes des protocoles restent très proches des solutions basiques non-tolérantes aux fautes. Ce qui montre qu'en pratique aussi la propriété demandée pour transformer des protocoles avec notre transformateur est plus simple à obtenir que l'auto-stabilisation ou la stabilisation instantanée. En outre, le protocole de parcours en profondeur que nous réalisons avec notre transformateur est non seulement trivial à écrire mais ses performances en font un compromis quasi idéal entre les protocoles à listes et à questions présentés dans le chapitre 5. Enfin, grâce à une propriété de comptage due à notre transformateur, nous montrons comment utiliser ce protocole de parcours pour résoudre l'exclusion mutuelle de manière instantanément stabilisante. En fait, cette propriété permet à tout processeur de détecter quand il détient un jeton unique dans le réseau. Ainsi, en utilisant cette méthode de détection simple, nous obtenons trivialement un protocole d'exclusion mutuelle instantanément stabilisant.

La perspective directe des travaux présentés dans cette thèse concerne notre transformateur. Pour le moment, il se restreint à rendre instantanément stabilisant les protocoles de service mono-initiateurs où les règles de décision apparaissent dans le code de l'initiateur uniquement. Bien que ce type de protocole permette de résoudre des problèmes fondamentaux de manière instantanément stabilisante, il est légitime de se demander s'il est possible de transformer de manière efficace (*i.e.*, sans utiliser de calculs d'état global du système), une classe plus large de protocoles. Par exemple, peut-on transformer efficacement en protocoles instantanément stabilisants des protocoles de service mono-initiateurs qui ne comportent pas de règles de décision ? des protocoles de service mono-initiateurs dont les règles de décision ne s'exécutent pas uniquement au niveau de l'initiateur ? ou des protocoles de service multi-initiateurs ?

Une autre perspective concerne la propriété *SSBB-Friendly* : la propriété que doit vérifier notre protocole initial pour être transformé automatiquement par notre protocole. Bien que cette propriété soit plus faible que la propriété de stabilisation (auto-stabilisation ou stabilisation instantanée), nous ne savons pas si cette propriété est minimale, *i.e.*, existe-t-il une propriété strictement induite permettant de transformer efficacement des protocoles de service mono-initiateurs où les règles de décision s'exécutent uniquement au niveau de l'initiateur en protocoles instantanément stabilisants ?

Enfin, démontrer la possibilité (ou l'impossibilité) d'écrire des protocoles de service instantanément stabilisants dans le modèle à passage de messages (un modèle plus proche de la réalité que le modèle à états) reste une question ouverte. La difficulté d'obtenir des solutions stabilisantes dans le modèle à messages est liées à la stabilisation des canaux de communication. Les travaux de Dolev *et al* ([DIM93, DIM91]) ainsi que ceux d'Afek et Brown ([AB93]) montrent que résoudre le problème de la communication point à point de manière déterministe et auto-stabilisante sans hypothèse sur la capacité des canaux nécessite un nombre d'états du système potentiellement infini. Il peut s'agir de la mémoire locale des processeurs qui devient infinie ou du nombre de messages dans les canaux qui ne cessent d'augmenter. Le fait de résoudre la communication point à point de manière auto-stabilisante n'est pas suffisant pour pouvoir construire des protocoles instantanément stabilisants : en effet, il faut que les processeurs soient capables de détecter le moment à partir duquel ils sont sûrs que les canaux sont stabilisés. Or, le fait que la stabilisation des canaux nécessite un nombre non borné d'états rend une telle détection difficilement envisageable. Donc, sans borne sur la capacité des canaux, nous pouvons raisonnablement conjecturer qu'il n'est pas possible de construire des protocoles instantanément stabilisants. En revanche, nous conjecturons qu'il est possible de construire des protocoles instantanément stabilisants dans le modèle à messages lorsque la capacité des canaux de communication est supposée bornée.

Troisième partie

Annexes

Annexe A

Théorie des graphes

A.1 Définitions de base

Définition A.1.1 (Graphe orienté) *Un graphe orienté G est un couple (V,E) où V est un ensemble de* sommets *et E un ensemble de couples, appelés* arcs*, de la forme (p,q) tel que p, $q \in V$. On notera n le cardinal de V et m le cardinal de E.*

Définition A.1.2 (Successeur et prédécesseur) *Soit $G=(V,E)$ un graphe orienté. q est dit* successeur *(respectivement* prédécesseur*) de p si et seulement si $(p,q) \in E$ (respectivement, $(q,p) \in E$). On note Γ_p^+ (respectivement, Γ_p^-) l'ensemble des successeurs (respectivement, prédécesseurs) de p.*

Définition A.1.3 (Degrés entrant et sortant) *Soit $G = (V,E)$ un graphe orienté. $\forall p \in V$, le* degré sortant *de p, noté Δ_p^+, est égal à $|\Gamma_p^+|$ (i.e., au nombre de sommets q tels que $(p,q) \in E$). Le* degré entrant *de p, noté Δ_p^-, est égal à $|\Gamma_p^-|$ (i.e., au nombre de sommets q tels que $(q,p) \in E$).*

Définition A.1.4 (Graphe non-orienté) *Un graphe non-orienté G est un couple (V, E) où V est un ensemble de* sommets *et E un ensemble de paires, appelés* arêtes*, de la forme $\{p,q\}$ (n.b. $\{p,q\} = \{q,p\}$) tel que p, $q \in V$ et $p \neq q$. $\forall \{p,q\} \in E$, p et q sont dit* incidents *de $\{p,q\}$. On notera n le cardinal de V et m le cardinal de E.*

Définition A.1.5 (Voisin) *Soit $G = (V,E)$ un graphe non-orienté. Deux sommets p et q sont dit* voisins *dans G dans si et seulement si $\{p,q\} \in E$. On notera Γ_p l'ensemble des voisins de p.*

Définition A.1.6 (Degré) *Soit $G = (V,E)$ un graphe non-orienté. Le* degré *de $p \in V$, noté Δ_p, est égal à $|\Gamma_p|$ (i.e., le nombre d'arêtes dont p est incident). Le degrée de G, noté Δ, est égal à* $\max(\{\Delta_p :: p \in V\})$.

Définition A.1.7 (Graphe partiel) *Soit $G = (V,E)$ un graphe. $G_P = (V_P,E_P)$ est un* graphe partiel *de G si et seulement si $V_P = V$ et $E_P \subseteq E$.*

Définition A.1.8 (Sous-graphe) *Soit $G = (V,E)$ un graphe. $G_S = (V_S,E_S)$ est un* sous-graphe *de G si et seulement si $V_S \subseteq V$ et $E_S = E \cap (V_S \times V_S)$.*

Définition A.1.9 (Chemin) *Soit $G = (V,E)$ un graphe non-orienté (respectivement, orienté). La suite p_1, p_2, ..., p_k est un* chemin[1] *de longueur $k-1$ dans G si et seulement si $\forall i$, $1 \leq i < k$, $\{p_i,p_{i+1}\} \in E$ (respectivement, $(p_i,p_{i+1}) \in E$).*

[1] Afin d'éviter toute confusion avec les chemins d'un graphe orienté, les chemins d'un graphe non-orienté sont aussi appelés *chaînes*.

A.2 Définitions avancées

Définition A.2.1 (Chemin élémentaire) *Soit $G = (V,E)$ un graphe. Soit $P = p_1, p_2, ..., p_k$ un* chemin *de G. P est dit* élémentaire *si et seulement si $\forall i, j, 1 \leq i < j \leq k, p_i \neq p_j$.*

Définition A.2.2 (Cycle) *Soit $G = (V,E)$ un graphe non-orienté. Soit $P = p_1, p_2, ..., p_k$ un* chemin *de G. P est un* cycle *si et seulement si $p_1, p_2, ..., p_{k-1}$ est* élémentaire *et $p_1 = p_k$.*

Définition A.2.3 (Circuit) *Soit $G = (V,E)$ un graphe orienté. Soit $P = p_1, p_2, ..., p_k$ un* chemin *de G. P est un* circuit *si et seulement si $p_1, p_2, ..., p_{k-1}$ est* élémentaire *et $p_1 = p_k$.*

Définition A.2.4 (Connexe) *Soit $G = (V,E)$ un graphe non-orienté. G est dit* connexe *si et seulement si $\forall p, q \in V$ tels que $p \neq q$, il existe un chemin dans G de p vers q.*

Définition A.2.5 (Composante connexe) *Soit $G=(V,E)$ un graphe non-orienté. $C = \{V_C,E_C\}$ est une* composante connexe *de G si et seulement si C est un sous-graphe connexe* maximal *de G (maximal signifie que $\forall V', V_C \subset V' \subseteq V$, le sous-graphe de G induit par V' n'est pas connexe).*

Définition A.2.6 (Plus court chemin) *Soit $G=(V,E)$ un graphe non-orienté connexe. Le* plus court chemin *entre deux sommets p et q dans G est un chemin de longueur k pour lequel il n'existe pas d'autre chemin de p vers q dans G ayant une longueur inférieur à k.*

Définition A.2.7 (Distance) *Soit $G = (V,E)$ un graphe non-orienté connexe. La* distance *entre deux sommets p et q dans G, notée $Dist(p, q)$, correspond à la longueur du plus court chemin entre p et q dans G.*

Définition A.2.8 (Diamètre) *Soit $G = (V,E)$ un graphe non-orienté connexe. Le* diamètre *de G, noté D, est égal à max($\{Dist(p, q) :: p,q \in V\}$).*

Définition A.2.9 (Arbre) *Tout graphe non-orienté connexe sans cycle est appellé* arbre. *On distingue deux types de sommets dans un arbre :*
- *les* feuilles *dont le degré est 1 ;*
- *les* noeuds internes *dont le degré est supérieur à 1.*

Définition A.2.10 (Arbre enraciné) *On dit qu'un arbre $T = (V_T,E_T)$ est* enraciné *lorsque l'on distingue l'un de ses sommets. On note $T(\mathrm{r})$ l'arbre T enraciné en* r *($\mathrm{r} \in V$). Le sommet* r *est alors appelé* racine *de T. Définir une racine dans un arbre permet de partiellement ordonner les sommets de cet arbre en fonction de leurs distances à la racine. Soit $T(\mathrm{r}) = (V_T,E_T)$ un arbre enraciné en* r. *La hauteur d'un sommet p dans $T(\mathrm{r})$, noté $h(p)$, est égale $Dist(p,\mathrm{r})$. La hauteur de l'arbre $T(\mathrm{r})$, noté H, est égal à* max($\{h(p) :: p \in V_T\}$). *$\forall p \in V_T \setminus \{\mathrm{r}\}$, on appelle* père *du sommet p, l'unique sommet q tel que $h(q) = h(p) - 1$ dans $T(\mathrm{r})$. Réciproquement, p est dit* fils *de q dans $T(\mathrm{r})$. Un sommet p_1 est dit* ancêtre *d'un sommet p_k dans $T(\mathrm{r})$ si et seulement si un chemin $p_1, ..., p_i, ..., p_k$ dans $T(\mathrm{r})$ tel que $\forall j, 1 \leq j < k$, p_j est le père de p_{j+1}. Réciproquement, p_k est dit* descendant *de p_k dans $T(\mathrm{r})$. On appelle* sous-arbre *de $T(\mathrm{r})$ enraciné en $p \in V_T$, noté $T(p)$, le sous-graphe de T induit par l'ensemble constitué de p et ses descendants dans T. Par commodité, nous noterons $q \in T(p)$ le fait que q appartienne au sous-arbre enraciné en p. Enfin, par abus de langage, nous pourrons noter $T(p) = \{p_1, p_2, ..., p_k\}$ où $\{p_1, p_2, ..., p_k\}$ désignera l'ensemble des processeurs contenus dans $T(p)$.*

Définition A.2.11 (Arbre couvrant) *Soit $G = (V,E)$ un graphe non-orienté connexe. $T = (V,E_T)$ est un* arbre couvrant *de G si et seulement si T est un graphe partiel connexe de G tel que $|E_T| = n - 1$.*

Définition A.2.12 (Arbre couvrant en profondeur d'abord) *Soit G=(V,E) un graphe non-orienté connexe. $T(\mathrm{r}) = (V,E_T)$ est un* arbre couvrant en profondeur d'abord *de G enraciné en* r *si et seulement si $T(\mathrm{r})$ vérifie les deux conditions suivantes :*

- *$T(\mathrm{r})$ est un arbre couvrant de G.*
- *Pour tout couple (p,q) de voisins de G, p est soit un ancêtre soit un descendant de q dans $T(\mathrm{r})$.*

Définition A.2.13 (Arbre couvrant en largeur) *Soit $G = (V,E)$ un graphe non-orienté connexe. $T(\mathrm{r}) = (V,E_T)$ est un* arbre couvrant en largeur *de G enraciné en* r *si et seulement si $T(\mathrm{r})$ vérifie les deux conditions suivantes :*

- *$T(\mathrm{r})$ est un arbre couvrant de G.*
- *$\forall p \in V$, la longueur du chemin élémentaire de p à* r *dans $T(\mathrm{r})$ est égale à la distance entre p et* r *dans G.*

Définition A.2.14 (Ensemble séparateur) *Soit $G = (V,E)$ un graphe non-orienté connexe. Le sous-ensemble de V, CS, est un* séparateur *de G si et seulement si le sous-graphe induit par $V \setminus CS$ n'est pas connexe.*

Définition A.2.15 (Point d'articulation) *Soit $G = (V,E)$ un graphe non-orienté connexe. $\forall p \in V$, p est un* point d'articulation *de G si et seulement si le sous-graphe induit par $V \setminus \{p\}$ n'est pas connexe.*

Définition A.2.16 (Isthme) *Soit $G = (V,E)$ un graphe non-orienté connexe. $\forall\ e \in E$, e est un* isthme *de G si et seulement si le graphe partiel $G' = (V,\ E \setminus \{e\})$ n'est pas connexe.*

Bibliographie

[AB93] Y Afek and GM Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7) :27–34, 1993.

[AB98] Y Afek and A Bremler. Self-stabilizing unidirectional network algorithms by power supply. *Chicago Journal of Theoretical Computer Science*, 1998 :Article 3, 1998.

[AG85] B Awerbuch and RG Gallager. Distributed BFS algorithms. In *FOCS 1985*, pages 250–256, 1985.

[AG94] A Arora and MG Gouda. Distributed reset. *IEEE Transactions on Computers*, 43 :1026–1038, 1994.

[Ahm88] S H Ahmad. Simple enumeration of minimal cutsets of acyclic directed graph. *IEEE Transactions on Reliability*, 37 :484–487, 1988.

[AK93] S Aggarwal and S Kutten. Time optimal self-stabilizing spanning tree algorithms. In *FSTTCS'93*, pages 400–410, 1993.

[AKM+93] B Awerbuch, S Kutten, Y Mansour, B Patt-Shamir, and G Varghese. Time optimal self-stabilizing synchronization. In *STOC93 Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 652–661, 1993.

[AKY90] Y Afek, S Kutten, and M Yung. Memory-efficient self-stabilization on general networks. In *WDAG90 Distributed Algorithms 4th International Workshop Proceedings, Springer-Verlag LNCS :486*, pages 15–28, 1990.

[APSV91] B Awerbuch, B Patt-Shamir, and G Varghese. Self-stabilization by local checking and correction. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.

[AU92] A. Aho and J. Ullman. *Concept fondamentaux de l'informatique*. Dunod, 1992.

[Awe85a] B Awerbuch. Complexity of network synchronization. *Journal of the Association of the Computing Machinery*, 32(4) :804–823, 1985.

[Awe85b] B Awerbuch. A new distributed depth-first-search algorithm. *Information Processing Letters*, 20 :147–150, 1985.

[AZ89] M Ahuja and Y Zhu. An efficient distributed algorithm for finding articulation points, bridges and biconnected components in asynchronous networks. In *9th Conference on Foundations of Software Technology and Theorical Computer Science, Banglore, India*, pages 99–108. LNCS 405, 1989.

[BB90] J Beauquier and B Bérard. *Systèmes d'exploitation*. McGraw-Hill, Paris, France, 1990.

[BCV03] L Blin, A Cournier, and V Villain. An improved snap-stabilizing PIF algorithm. In *DSN SSS'03 Workshop : Sixth Symposium on Self-Stabilizing Systems (SSS'03)*, pages 199–214. LNCS 2704, 2003.

[BDPV99a] A Bui, AK Datta, F Petit, and V Villain. Snap-stabilizing PIF algorithm in tree networks without sense of direction. In *SIROCCO'99, The 6th International Colloquium On Structural Information and Communication Complexity Proceedings*, pages 32–46. Carleton University Press, 1999.

[BDPV99b] A Bui, AK Datta, F Petit, and V Villain. Space optimal PIF algorithm : Self-stabilizing with no extra space. In *IPCCC'99, IEEE International Performance, Computing, and Communications Conference*, pages 20–26. IEEE Computer Society Press, 1999.

[BDPV99c] A Bui, AK Datta, F Petit, and V Villain. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the Fourth Workshop on Self-Stabilizing Systems*, pages 78–85, Austin, Texas, USA, June 1999. IEEE Computer Society Press.

[BDV05] D Bein, AK Datta, and V Villain. Snap-stabilizing optimal binary search tree. In *Seventh International Symposium on Self-Stabilizing Systems (SSS'05)*, pages 1–17, Barcelona, Spain, 2005. LNCS 3764.

[Ber83] C Berge. *Graphes et hypergraphes*. Bordas, 1983.

[CD94] Z Collin and S Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49(6) :297–301, 1994.

[CDPV01a] A Cournier, AK Datta, F Petit, and V Villain. Optimal snap-stabilizing PIF in unoriented trees. In *OPODIS 2001, Fifth International Conference On Principles Of Distributed Systems*, 2001.

[CDPV01b] A Cournier, AK Datta, F Petit, and V Villain. Self-stabilizing PIF algorithm in arbitrary rooted networks. In *21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 91–98. IEEE Computer Society Press, 2001.

[CDPV02] A Cournier, AK Datta, F Petit, and V Villain. Snap-stabilizing PIF algorithm in arbitrary rooted networks. In *22st International Conference on Distributed Computing Systems (ICDCS-22)*, pages 199–206. IEEE Computer Society Press, 2002.

[CDPV03] A Cournier, AK Datta, F Petit, and V Villain. Enabling snap-stabilization. In *23th International Conference on Distributed Computing Systems (ICDCS 2003)*, pages 12–19, Providence, Rhode Island USA, May 19-22 2003. IEEE Computer Society Press.

[CDPV04] A Cournier, S Devismes, F Petit, and V Villain. Snap-stabilizing depth-first search on arbitrary networks. In *OPODIS'04, International Conference On Principles Of Distributed Systems Proceedings*, pages 267–282, Grenoble, France, 2004. LNCS 3544.

[CDPV06] A Cournier, S Devismes, F Petit, and V Villain. Snap-Stabilizing Depth-First Search on Arbitrary Networks. *The Computer Journal*, 49(3) :268–280, 2006.

[CDV05a] A Cournier, S Devismes, and V Villain. Snap-stabilizing detection of cutsets. In *HIPC 2005, 12th Annual IEEE Conference on High Performance Computing*, pages 488–497. LNCS 3769, 2005.

[CDV05b] A Cournier, S Devismes, and V Villain. A snap-stabilizing DFS with a lower space requirement. In *Seventh International Symposium on Self-Stabilizing Systems (SSS'05)*, pages 33–47, Barcelona, Spain, 2005. LNCS 3764.

[CDV06a] A Cournier, S Devismes, and V Villain. From Self- to Snap- Stabilization. In *8th International Symposium on Self-Stabilization, Safety, and Security (SSS'06)*, pages 199–213, Dallas, USA, 2006. LNCS 4280.

[CDV06b] A Cournier, S Devismes, and V Villain. Snap-stabilizing PIF and useless computations. In *The Twelfth International Conference on Parallel and Distributed Systems (ICPADS'06)*, volume 1, pages 39–46, Minneapolis, USA, 2006. IEEE Computer Society Press P2612.

[CDV06c] A Cournier, S Devismes, and V Villain. Snap-stabilizing PIF and useless computations. Technical Report LaRIA-2006-04, LaRIA, CNRS FRE 2733, 2006. Available at www.laria.u-picardie.fr/∼devismes/LaRIA-2006-04.pdf.

[Cha82] EJH Chang. Echo algorithms : depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, SE-8 :391–401, 1982.

[Cha99a] P Chaudhuri. A note on self-stabilizing articulation point detection. *Journal of Systems Architecture*, 45(14) :1249–1252, 1999.

[Cha99b] P Chaudhuri. An $o(n^2)$ self-stabilizing algorithm for computing bridge-connected components. *Computing*, 62 :55–67, 1999.

[Che83] T Cheung. Graph traversal techniques and maximum flow problem in distributed computation. *IEEE Transactions on Software Engineering*, SE-9(4) :504–512, 1983.

[Cid88] I Cidon. Yet another distributed depth-first-search algorithm. *Information Processing Letters*, 26 :301–305, 1988.

[CM82] K. M. Chandy and J. Misra. Distributed computation on graphs : Shortest path algorithms. *Communications of the ACM*, 25(11), 1982.

[CMH83] K. M. Chandy, J. Misra, and L. M. Haas. Distributed deadlock detection. *ACM Transactions on Computing Systems*, 1(2) :144–156, 1983.

[Cou02] A Cournier. One dollar questions. Preliminary work, January 2002.

[CR79] EJH Chang and R Roberts. An improved algorithm for decentralized extrema finding in circular arrangements of processes. *Communications of the ACM*, 22 :281–283, 1979.

[CT91] TD Chandra and S Toueg. Unreliable failure detectors for asynchronous systems. In *PODC91, The Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 325–340, 1991.

[CYH91] NS Chen, HP Yu, and ST Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39 :147–151, 1991.

[Dev05] S Devismes. A silent self-stabilizing algorithm for finding cut-nodes and bridges. *Parallel Processing Letters*, 15 :183–198, 2005.

[DGPV00] Ajoy Kumar Datta, Shivashankar Gurumurthy, Franck Petit, and Vincent Villain. Self-stabilizing network orientation algorithms in arbitrary rooted networks. In *International Conference on Distributed Computing Systems*, pages 576–583, 2000.

[DGS96] S. Dolev, MG Gouda, and M Schneider. Memory requirements for silent stabilization. In *PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 27–34, 1996.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8,9 :569, September 1965.

[Dij74] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17 :643–644, 1974.

[DIM91] S Dolev, A Israeli, and S Moran. Resource bounds for self stabilizing message driven protocols. In *PODC91, the tenth annual ACM symposium on Principles Of Distributed Computing*, pages 281–293, 1991.

[DIM93] S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7 :3–16, 1993.

[DIM97] S Dolev, A Israeli, and S Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4) :424–440, 1997.

[DJPV00] AK Datta, C Johnen, F Petit, and V Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. *Distributed Computing*, 13(4) :207–218, 2000.

[DS80] E. W. Dijkstra and C. S. Scholten. Termination detection from diffusing computations. *Information Processing Letters*, 11(1) :1–4, 1980.

[DT01] B Ducourthial and S Tixeuil. Self-stabilization with r-operators. *Distributed Computing*, 14(3) :147–162, July 2001.

[FD94] M Flatebo and AK Datta. Two-state self-stabilizing algorithms for token rings. *IEEE Transactions on Software Engineering*, 20 :500–504, 1994.

[FL99] N S Fard and T H Lee. Cutset enumeration of network systems with link and node failure. *Reliability Engineering and System Safety*, 65 :141–146, 1999.

[FLP85] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association of the Computing Machinery*, 32(2) :374–382, 1985.

[Fra80] N. Francez. Distributed termination. *ACM Transactions on Programming Languages and Systems*, 2(1) :42–55, 1980.

[GGHP00] S Ghosh, A Gupta, T Herman, and SV Pemmaraju. Fault-containing self-stabilizing distributed protocols. Technical Report 00-01, Department of Computer Science, University of Iowa, 2000.

[GHS83] R. G. Gallager, P. A. Humbet, and P. M. A. Spira. A distributed algorithm for minimum weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5 :67–77, 1983.

[HC92] ST Huang and NS Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, pages 41 :109–117, 1992.

[HC93] ST Huang and NS Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7 :61–66, 1993.

[Her91a] T Herman. *Adaptivity through distributed convergence*. PhD thesis, University of Texas at Austin, Departement of Computer Science, 1991.

[Her91b] T Herman. Binary self-stabilization in distributed systems. *Information Processing Letters*, 40 :153–159, 1991.

[Her92] T Herman. Self-stabilization : ramdomness to reduce space. *Information Processing Letters*, 6 :95–98, 1992.

[JABD97] C Johnen, C Alari, J Beauquier, and AK Datta. Self-stabilizing depth-first token passing on rooted networks. In *WDAG97 Distributed Algorithms 11th International Workshop Proceedings, Springer-Verlag LNCS :1320*, pages 260–274, Saarbrücken, Germany, September 24-26 1997. Springer-Verlag.

[JADT02] Colette Johnen, Luc Alima, Ajoy K. Datta, and Sébastien Tixeuil. Optimal snap-stabilizing neighborhood synchronizer in tree networks. *Parallel Processing Letters*, 12(3-4) :327–340, 2002.

[JB95] Colette Johnen and Joffroy Beauquier. Space-efficient distributed self-stabilizing depth-first token circulation. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 4.1–4.15, Las Vegas (UNLV), USA, May 28-29 1995. Chicago Journal of Theoretical Computer Science.

[Joh97] C Johnen. Memory efficient, self-stabilizing algorithm to construct BFS spanning trees. In *PODC97 Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, page 288, 1997.

[JSYZ04] Joxan Jaffar, Andrew E. Santosa, Roland H. C. Yap, and Kenny Q. Zhu. Scalable distributed depth-first search with greedy work stealing. In *ICTAI '04 : Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'04)*, pages 98–103, Washington, DC, USA, 2004. IEEE Computer Society.

[Kar99] Mehmet Hakan Karaata. A self-stabilizing algorithm for finding articulation points. *International Journal of Foundations of Computer Science*, 10(1) :33–46, 1999.

[Kar00] D R Karger. Minimum cuts in near-linear time. *Journal of the ACM*, 47 :46–76, 2000.

[KC99] M Hakan Karaata and P Chaudhuri. A self-stabilizing algorithm for bridge finding. *Distributed Computing*, 12(1) :47–53, 1999.

[KP93] S Katz and KJ Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7 :17–26, 1993.

[Kra87] S. Krakowiak. *Principes des systèmes d'exploitation des ordinateurs*. Dunod, 1987.

[Kru79] HSM Kruijer. Self-stabilization (in spite of distributed control) in tree-structured systems. *Information Processing Letters*, 8 :91–95, 1979.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7) :558–564, 1978.

[LL77] G Le Lann. Distributed systems : towards a formal approach. In *Information Processing'77*, pages 155–160, 1977.

[LY87] T. H. Lai and T. H. Yang. On distributed snapshots. *Information Processing Letters*, 25 :153–158, 1987.

[Lyn96] N Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[Mat87] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3) :161–175, 1987.

[Mat88] F Mattern. Virtual time and global states of distributed systems. In *Proceedings of International Workshop on Parallel and Distributed System*, pages 215–226, 1988.

[MM79] D. Menasce and R. Muntz. Locking and deadlock detection in distributed databases. *Communications of the ACM*, 21, 1979.

[MM88] D. P. Mitchell and M. J. Merritt. A distributed algorithm for deadlock detection and resolution. In *Proceedings of the Third Annual ACM Symposium pn Principles of Distributed Computing*, pages 215–226, 1988.

[MW87] S. Moran and Y. Wolfstahl. Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26 :145–151, 1987.

[NMFS02] N. Nellari, D. Maric, Stavros C. Farantos, and Stamatis Stamatiadis. Report on grid enabling technologies in the context of the enacts collaboration. Technical Report 2002-12-31, SCSC, 2002.

[Nol02] Florent Nolot. *Stabilisation des horloges de phases dans les systèmes distribués*. PhD thesis, Université de Picardie Jules Verne, LaRIA, Amiens, 2002.

[NT87] M. Naimi and M. Trehel. How to detect and regenerate the token in the log(N) distributed algorithm for mutual exclusion. In *Proceedings of 7th IEEE International Conference on Distributed Computing Systems*, pages 371–375, 1987.

[Pat71] K Paton. An algorithm for blocks and cutnodes of a graph. *Communications of the ACM*, 37 :468–475, 1971.

[PB83] J S Provan and M O Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM Journal of Computing*, 12 :777–788, 1983.

[Pet98] Franck Petit. *Efficacité et simplicité dans les algorithmes distribués auto-stabilisants de parcours en profondeur de jeton*. PhD thesis, Université de Picardie Jules Verne, LaRIA, Amiens, 1998.

[Pet01] F Petit. Fast self-stabilizing depth-first token circulation. In *Proceedings of the Fifth Workshop on Self-Stabilizing Systems, Lisbonne (Portugal)*, pages 200–215. LNCS 2194, October 2001.

[Pil05] Laurence Pilard. *Observer la stabilisation*. PhD thesis, Université Paris XI, Orsay, 2005.

[PTMH91] J Parks, N Tokura, T Masuzawa, and K Hagihara. Efficient distributed algorithms solving problems about the connectivity of network. *Systems and Computers in Japan*, 22 :1–16, 1991.

[PV97] F Petit and V Villain. Color optimal self-stabilizing depth-first token circulation. In *I-SPAN'97, Third International Symposium on Parallel Architectures, Algorithms and Networks Proceedings*, pages 317–323. IEEE Computer Society Press, 1997.

[PV99] F Petit and V Villain. Time and space optimality of distributed depth-first token circulation algorithms. In *Proceedings of DIMACS Workshop on Distributed Data and Structures*, pages 91–106, Princeton, USA, May 10-11 1999. Carleton University Press.

[PV03] F Petit and V Villain. Optimal snap-stabilizing depth-first token circulation in tree networks. Technical Report LaRIA-2003-12, LaRIA, CNRS FRE 2733, 2003.

[RA81] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion. *Communications of the ACM*, 24(1) :9–17, 1981.

[Rai82] S Rai. A cutset approach to reliability evaluation in communication networks. *IEEE Transactions on Reliability*, 31 :428–431, 1982.

[Ray85] M Raynal. *Algorithmes Distribués et Protocoles*. Eyrolles, 1985.

[Seg83] A Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29 :23–35, 1983.

[SK92] A P Spraque and K H Kulkarni. Optimal parallel algorithms for finding cuter vertices and bridges of internal graphs. *Information Processing Letters*, 42 :229–234, 1992.

[SS92] S Sur and PK Srimani. A self-stabilizing distributed algorithm to construct BFS spanning trees on a symetric graph. *Parallel Processing Letters*, pages 2(2/3) :171–179, 1992.

[Taj77] W D Tajibnapis. A correctness proof of a topology information maintenance protocol for a distributed computer network. *Communications of the ACM*, 20(7) :477–485, 1977.

[Tar72] Robert E Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1 :No 2, june 1972.

[Tel01] G Tel. *Introduction to distributed algorithms*. Cambridge University Press, Cambridge, UK, Second edition 2001.

[Tou80] S. Toueg. An all-pairs shortest-path distributed algorithm. Technical Report RC 8327, IBM T. J. Watson Research Center, Yorktown, NY 10598, 1980.

[Var93] G Varghese. Self-stabilization by local checking and correction (Ph.D. thesis). Technical Report MIT/LCS/TR-583, MIT, 1993.

[Vil02] V Villain. Snap-stabilization versus self-stabilization. Journées internationales sur l'auto-stabilisation, October 2002. CIRM, Luminy (France).

[vLT87] Jan van Leeuwen and Richard B. Tan. Interval routing. *Comput. J.*, 30(4) :298–307, 1987.

[WSJ90] D E Whited, D R Shier, and J P Jarvis. Reliability computations for planar networks. *OSRA Journal of Computing*, 2(1) :46–60, 1990.

Quelques Contributions à la Stabilisation Instantanée

Résumé

Dans cette thèse, nous nous sommes intéressés au concept de *stabilisation instantanée*. Ainsi, nous avons tout d'abord proposé deux solutions instantanément stabilisantes au problème de *parcours en profondeur* pour des réseaux enracinés quelconques. Ces deux protocoles sont écrits dans le modèle à états et fonctionnent sous l'hypothèse d'un démon distribué inéquitable : le démon le plus général du modèle. Le premier est basé sur des listes d'identités. Le second utilise un principe de question/réponse pour remplacer les listes d'identités. Nous proposons ensuite deux applications instantanément stabilisantes obtenues à partir de nos deux protocoles de parcours en profondeur. Ces deux applications évaluent des propriétés globales sur le réseau. La première application permet de marquer les *points d'articulation* et les *isthmes* du réseau. La seconde application permet d'évaluer si un ensemble donné est un *ensemble séparateur* du réseau. Enfin, dans une dernière partie, nous adoptons une approche plus générale en étudiant un protocole efficace permettant de transformer semi-automatiquement des protocoles de service mono-initiateurs en protocoles instantanément stabilisants. Un protocole de parcours en profondeur et un protocole de construction d'arbre en largeur illustrent la facilité avec laquelle nous pouvons rendre instantanément stabilisants ce type protocole grâce à notre transformateur. Le protocole de parcours en profondeur est non seulement trivial à écrire mais les performances obtenues en font un compromis quasi idéal entre les protocoles à listes et à questions présentés précédemment. Enfin, grâce à une propriété de comptage due à notre transformateur, nous montrerons comment utiliser ce protocole de parcours pour résoudre en quelques lignes l'*exclusion mutuelle* de manière instantanément stabilisante.

Mots-clés : systèmes distribués, tolérance aux fautes, stabilisation instantanée, auto-stabilisation, transformateur, parcours en profondeur, exclusion mutuelle.

Some Contributions to Snap-Stabilization

Abstract

In this PhD thesis, we are interested in the notion of *snap-stabilization*. Thus, we first proposed two solutions to the *depth-first search* problem in arbitrary rooted networks. These two protocols are written in the state model and work under a distributed unfair daemon : the weakest scheduling assumption of the model. The first one is based on IDs lists. The other one uses a question mechanism instead of the IDs lists. We then propose two snap-stabilizing applications obtained with the previously presented depth-first search protocols. These two applications allow us to evaluate some global network properties. Actually, the first application marks the *cutnodes* and the *bridges* of the network. The second application allows us to evaluate if a given set of processors is a *cutset* of the network. Finally, in the last part, we generalize our approach by studying an efficient semi-automatic transformer that snap-stabilizes service protocols with a unique initiator. A depth-first search protocol as well as a breath-first spanning tree construction protocol show how our method is easy. Our depth-first search protocol is not only easy to write but its complexity is a trade-off between the list-based and the question-based solutions. Finally, using a counting property of our transformer, we show how to use this depth-first search protocol to solve in few lines the *mutual exclusion problem* in a snap-stabilizing manner.

Keywords : distributed systems, fault-tolerance, snap-stabilization, self-stabilization, transformer, depth-first search, mutual exclusion.