# Table des matières

# Introduction : il était une fois...

*"La probabilité qu'a un ordinateur de tomber en panne au moment où l'utilisateur en a vraiment besoin est proportionnelle à l'urgence du travail devant être accompli"*

Murphy, $7^{eme}$ loi

## 0.1 Les schtroumpfs, mythe ou réalité

La première apparition publique des schtroumpfs date de 1958. Petit lutin à peau bleue et à bonnet blanc, le schtroumpf est un animal grégaire nichant le plus souvent dans un champignon.

Leur organisation sociale présente très peu de défauts. Chaque schtroumpf a un rôle clairement défini qu'il est généralement heureux de remplir. Tout au plus arrive-t-il épisodiquement que l'un d'entre eux modifie son statut en changeant de spécialité, mais cela ne prête en général pas à conséquence.

**Le problème** Deux grains de sable viennent perturber ce système politique presque parfait. Le premier fait son apparition dans [Pey76] sous forme d'une entité particulièrement attractive qui provoque la convergence des pensées de tous les schtroumpfs vers un objet unique : elle-même. Par malheur, cette entité (surnommée "la schtroumpfette") n'existe qu'en un seul exemplaire. Malgré la pénurie, les schtroumpfs sont trop civilisés pour se livrer à une guerre ouverte. Mais ils ne s'en trouvent pas moins confrontés à un serieux problème : une schtroumpfette, cent schtroumpfs...

Dans [Pey78], il est fait mention qu'une des activités préférées des schtroumpfs est d'assister à un coucher de soleil. Dès lors, assister à un coucher de soleil *avec* la schtroumpfette devient une consécration. Reste à cette dernière la difficile tâche de choisir son accompagnateur.

**Première solution :** Pour lui permettre un choix juste et équitable, le grand schtroumpf (une sorte de dictateur éclairé auquel tous obéîssent) établit une tradition. Les soirs de lune bleue, tous les schtroumpfs font une grande ronde autour de la schtroumpfette. A l'un d'entre eux, il est remis la "baguette de l'orateur". L'heureux possesseur de cette baguette dispose d'une minute pour lire un poème, chanter une chanson ou faire une quelconque déclaration susceptible d'émouvoir la schtroumpfette. Ensuite, l'orateur /

chanteur / déclarateur passe la baguette à son voisin de gauche qui peut également tenter sa chance pendant une minute. Et ainsi de suite jusqu'à ce que la schtroumpfette choisisse. Cette dernière étant par nature assez indécise, il n'est pas rare que le même schtroumpf puisse s'exprimer plusieurs fois. Mais les schtroumpfs sont patient...

Le vainqueur gagne tous les couchers de soleil jusqu'à la prochaine lune bleue.

**Le deuxième grain de sable.** Surnommé "schtroumpf farceur", il est omniprésent dans toute l'histoire des schtroumpfs. Sous son air débonnaire et guilleret, ce psycho-parnoïaque maniacodépressif dissimule une tendance compulsive et irrépressible au terrorisme systèmatique. Ses innombrables bombes font presque partie integrante de la vie des autres schtroumpfs.

Cela ne serait que demi mal s'il s'en tenait là. Mais sa nature perverse le pousse à régulièrement modifier son comportement. Un jour il pose une bombe, un autre jour il offre un gâteau, un troisième il tombe lui-même dans son propre piège... Bref, le schtroumpf farceur est problématique parce que complètement imprévisible.

**La lune bleue.** Même lors de la ronde de la lune bleue, moment considéré par beaucoup de schtroumpfs comme étant sacré, il se livre à ses continuelles farces. Régulièrement, il fabrique des imitations de la baguette de l'orateur et les distribue mesquinement avant que la cérémonie ne commence. Plusieurs schtroumpfs, tous se croyant orateurs, se mettent alors à parler simultanément. Naturellement, aucun ne veut se taire de peur de perdre son temps de parole. La schtroumpfette, centre de la cacophonie, ne comprend plus ce qu'on lui dit et ne peut plus choisir.

**Solution auto-stabilisante** Comme toujours, le grand schtroumpf trouva une solution. Il eut l'idée d'introduire deux types de baguettes : des rouges et des vertes. "La rouge, déclara-t-il, donne un temps de parole d'une minute, alors que la verte ne laisse que trente secondes. Et si un schtroumpf reçoit une baguette alors qu'il en a déjà une, il doit en jeter une et ne conserver que l'autre". Ainsi fut-il fait. Les schtroumpfs acceptèrent ce nouveau règlement avec complaisance.

De son côté, le grand schtroumpf décida qu'indépendamment de la couleur des baguettes qu'il recevrait, il transmettrait une fois sur deux une baguette rouge, une fois sur deux une baguette verte.

**Victoire !** Curieusement, en appliquant ces nouvelles règles, on constata une nette amélioration du déroulement de la cérémonie. Quand le schtroumpf farceur mettait en jeu plusieurs baguettes, les baguettes vertes circulant plus vite que les rouges, les schtroumpfs orateurs recevaient de temps en temps une deuxième baguette. Ils en jetaient alors une et progressivement, les baguettes excédentaires étaient éliminées. Bien sûr, le retour à la normale pouvait être long, mais les schtroumpfs sont patients...

**Solution rapide.** Pas si patients que ça, finalement... A tel point que l'un d'entre eux suggéra de ne plus transmettre systèmatiquement les baguettes dans le même sens. Après avoir parlé, un orateur devait regarder si un de ses voisins avait lui aussi une

baguette. Le cas échéant, plutôt que de transmettre systèmatiquement sa baguette au schtroumpf de gauche, il devait la transmettre de sorte qu'elle rattrape le plus vite possible une autre baguette. Bien sûr, en cas de baguette unique, la règle "transmettre à gauche" demeurait valable.

Sans le savoir, les schtroumpfs venaient de mettre en place un algorithme auto-stabilisant et un algorithme proportionnel.

## 0.2 Informatique et schtroumpferie

L'histoire précédente n'est qu'un exemple ludique illustrant trois des principaux concepts sur lesquels repose cette thèse : les algorithmes répartis, l'auto-stabilisation et la stabilisation proportionnelle.

En langage informatique, chaque schtroumpf est assimilable à un processeur. La ronde des schtroumpfs est un réseau. Les instructions qu'ils exécutent sont les codes programmes des processeurs. L'ensemble des schtroumpfs exécutant leur code illustre le fonctionnement d'un algorithme réparti.

Le problème qu'ils doivent résoudre est connu sous le nom "d'exclusion mutuelle" : plusieurs processeurs veulent accéder à une même ressource (l'oreille de la schtroumpfette) et ne peuvent le faire que un par un.

La première solution proposé est une solution répartie classique : Pour le résoudre, un protocole est mis en place (la première solution). Il fonctionne assez bien. Si tous les processeurs suivent leurs intructions et qu'aucune perturbation n'a lieu, il assure que tous les processeurs pourront avoir accès à la ressource.

La "mesquinerie" des ordinateurs n'a pas encore été établie de manière formelle et les célèbres lois de Murphy ne sont que des conjonctures. Néanmoins, il arrive fréquemment que les réseaux informatiques soient sujets à des perturbations. Elles sont ici représentées par la distribution de fausses baguettes, œuvre du schroumpf farceur[1].

Qu'elles frappent le réseau des schtroumpfs ou celui de systèmes informatiques plus classiques, les pannes ont bien souvent un effet fatal sur la bonne marche des opérations. Ainsi, la schtroumpfette n'entend plus les messages qui lui sont adressés.

L'auto-stabilisation est une propriété des algorithmes répartis leur permettant de corriger automatiquement un certain type de panne. Dans notre exemple, la solution auto-stabilisante permet au système de revenir progressivement vers une situation où un seule baguette subsiste[2]. Ce faisant, elle permet a l'algorithme de retrouver un comportement correct moyennant une petite phase transitoire où des perturbations sont encore présentes.

Diminuer la longueur de cette phase transitoire est le principal objectif des rafinements de l'auto-stabilisation que nous présentons ici. En particulier, les algorithmes stabilisants proportionnels ont la propriété de retrouver un comportement correct en un temps proportionnel à la taille de la panne.

1. Schroumpf farceur a qui nous présentons toutes nos excuses pour les qualificatifs dont nous l'avons affublé...

2. La preuve d'un algorithme sensiblement identique à celui-là est donné au chapitre 7.

## 0.3 Un peu d'histoire...

Nos travaux se placent dans le cadre général des études liées à l'auto-stabilisation.

### 0.3.1 L'auto-stabilisation

L'auto-stabilisation a été introduite par Dijkstra dans un article de 1974 [Dij74]. Pendant neuf ans, ce résultat resta marginal jusqu'à une intervention de Leslie Lamport [Lam83] qui fit remarquer l'importance et la puissance potentielle que présentait l'auto-stabilisation. Depuis, l'intérêt pour le domaine ne fait que croitre. [Sch93, FDG94, Tel94, HW95] présentent des "survey" de l'auto-stalisation.

### 0.3.2 Raffinements de l'auto-stabilisation

Le principal inconvénient des algorithmes auto-stabilisants est leur long temps de convergence. D'où l'idée naturelle de concevoir des algorithmes basés sur un concept proche de l'auto-stabilisation (pour en conserver les avantages) tout en proposant des temps de convergence moindre que leur équivalent auto-stabilisant. C'est l'objectif de ce que nous appelons "les raffinements de l'auto-stabilisation".

L'apparition de la $k$-stabilisation et des algorithmes proportionnels est résolument moderne. Jusqu'à une époque très récente, les seuls algorithmes proportionnels proposés résolvaient uniquement des problèmes statiques. Le concept apparaît pour la première fois dans [KP95]. Kutten et Patt-Shamir proposent un algorithme $k$-proportionnel permettant de résoudre le problème du bit persistant [KP97] alors que Afek et Dolev présente une méthode de transformation automatique dans [AD97]. En 1999, un algorithme à la fois $k$-stabilisant, auto-stabilisant et anti-corruption résoud pour la première fois un problème dynamique (l'exclusion mutuelle) [BGK99]. Puis, dans [Gen00], le même problème est résolu dans le cas synchrone de manière auto-stabilisante et proportionnelle.

### 0.3.3 L'exclusion mutuelle

Tout au long de cette thèse, nous allons illustrer les différentes techniques permettant d'obtenir des algorithmes proportionnels et anti-corruption par des exemples. Principalement, nous considèrerons le problème de l'exclusion mutuelle.

Les travaux sur le sujet sont innombrables. Pour mémoire, le premier algorithme d'exclusion mutuelle fut présenté par E. W. Dijkstra dans [Dij65]. M. Raynal leur consacre tout un livre ([Ray86]). Dans le cadre auto-stabilisant, c'est un problème fondamental qui a fait l'objet de très nombreuses études ([Dij74, GH96, FD94, BD95, BP89, IJ90, Her90, BCD95, DGT00]).

L'exclusion mutuelle auto-stabilisante a le même point de départ que l'auto-stabilisation puisque l'article de Dijkstra[Dij74] présente un algorithme d'exclusion mutuelle asynchrone sur un anneau semi-uniforme. Cet algorithme stabilise en $O(n^2)$ et utilise $O(n)$ états par processeur. Toujours sur un anneau semi-uniforme avec un démon asynchrone, M.G. Gouda et F.F. Haddix optimisent le nombre d'états par processeurs, au détriment toutefois du temps de stabilisation [GH96].

| **Problème traité** | **Topologie** | **Propriété** | **Démon** | **Coût mémoire** | **Temps de stabilisation** | **Réf.** |
|---|---|---|---|---|---|---|
| Exclusion mutuelle | Anneau semi-uniforme | Auto-stabilisant | Réparti Non-équitable | $O(log(n))$ | $O(n^2))$ | [Dij74] |
| Exclusion mutuelle | Anneau semi-uniforme | Auto-stabilisant | Réparti Non-équitable | 3 bits | $O(n^3))$ | [GH96] |
| Exclusion | Anneau uniforme | Impossible | Quelconque | | | |
| Exclusion mutuelle | Anneau uniforme de taille première | Auto-stabilisant | Réparti Non-équitable | $n^2/log(n)$ | | [BP89] |
| Exclusion mutuelle | Anneau uniforme | Auto-stabilisant | Réparti Probabiliste | | | [BCD95] |
| Bit persistant | Graphe uniforme | K-proportionnel quelconque | Réparti (avec $k < n/2$)) | $n^2$ | $O(f)$ | [KP97] |
| | | | | | | |
| Exclusion mutuelle | Anneau semi-uniforme | Auto-stabilisant Anti-corruption | Réparti Non-équitable | $O(log(n))$ | $O(n + k^2))$ | [BGK99] |
| Exclusion mutuelle | Anneau semi-uniforme | $k$-proportionnel impossible | Réparti Non-équitable | | | [BGK99] |
| Exclusion mutuelle | Anneau semi-uniforme | Auto-stabilisant | Synchrone | 1 bit | $O(n)$ | [Gen00] |
| Exclusion mutuelle | Anneau semi-uniforme | Proportionnel | Synchrone | $O(log(n))$ | $O(f^3 Log(f))$ | [Gen00] |
| Exclusion mutuelle | Anneau uniforme | $k$-proportionnel | Réparti Non-équitable | $O(log(k))$ | $O(f)$ | [Gen01] |

TAB. 1 – *Quelques algorithmes tolérant les défaillances transitoires*

L'hypothèse semi-uniforme est indispensable puisque le problème est insoluble de manière déterministe sur un anneau uniforme quelconque. Parmi les diverses solutions présentées sur un anneau uniforme, certaines sont fonction de la taille de l'anneau (ce qui implique que les processeurs en ont connaissance, comme dans [BP89]), d'autres sont probabilistes ([Her90, BCD95, KY97, DGT00]).

Le tableau 1 recense quelques algorithmes tolérant les défaillances transitoires. La deuxième moitié du tableau résume les différents résultats présentés dans cette thèse.

Le coût mémoire et le temps de convergence sont généralement fonction de certains paramètres, généralement liés à la topologie du système réparti pour lequel l'algorithme a été conçu. $n$ est la taille du graphe. $k$ est une constante à fixer lors de l'implantation de l'algorithme. $f$ est le nombre effectif de fautes (dans le cadre de la stabilisation proportionnelle ou $k$-proportionnelle).

## 0.4 Objectif de la thèse

L'objectif de cette thèse est de développer et synthétiser un certain nombre de concepts dérivés de l'auto-stabilisation.

Le chapitre 1, **Présentation du domaine**, introduit les algorithmes répartis. Il

définit différentes hypothèses permettant de modéliser un réseau.

Le chapitre 2, **Modèle**, fournit un cadre formel permettant de présenter des algorithmes répartis et de démontrer leurs propriétés. Tous les concepts abordés au premier chapitre sont modélisés et des définitions précises de l'auto-stabilisation et de ses raffinements y sont données. Diverses méthodes de démonstration utilisées pour prouver les propriétés des algorithmes sont exposées. Enfin, il présente plus spécifiquement les concepts liés à l'exclusion mutuelle sur un anneau.

Le chapitre 3 propose une étude de l'algorithme **d'Eclusion mutuelle de Dijkstra**. Après avoir présenté l'algorithme, nous étudions en détail sa complexité au pire.

Le chapitre 4 présente un résultat **d'impossibilité** : il n'existe pas d'algorithme d'exclusion mutuelle proportionnel sous un démon centralisé.

Les chapitre 5 et 6 proposent des techniques permettant d'améliorer les algorithmes auto-stabilisants. Appliqué à l'exclusion mutuelle, elles permettent d'obtenir un **algorithme anti-corruption** et ayant un temps de **convergence plus court** que celui de Dijkstra dans le cas d'un faible nombre de fautes. Ses techniques et les algorithmes les illustrant ont été publiées dans [BGK98, BGK99].

Un démon centralisé ne permettant pas aux algorithmes d'être proportionnel, les chapitres 7 et 8 présentent des algorithmes **d'exlusion mutuelle sous un démon synchrone**. Le premier, auto-stabilisant et optimal en espace mémoire, sert de base au second qui est lui **proportionnel**. Ses résultats ont été publiés dans [Gen00].

Enfin, le chapitre 9 montre tout l'interêt que représente le concept de stabilisation proportionnelle puisqu'il permet de **résoudre des problèmes insolubles** dans le cadre classique de l'auto-stabilisation.

# Chapitre 1

# Présentation générale du domaine

Dans ce chapitre, nous donnons un aperçu informel des systèmes répartis et des différents travaux qui s'y rapportent. Après avoir étudié certaines de leurs propriétés et leurs principales différences avec les algorithmes centralisés, nous présentons les caractéristiques générales de l'auto-stabilisation et des divers concepts qui en ont dérivé comme la $k$-stabilisation, la stabilisation proportionnelle ou la stabilisation anti-corruption. Puis nous étudions les hypothèses de réseau classiquement utilisées pour modéliser les algorithmes stabilisants. Enfin, à l'aide de tous ces concepts, nous examinons brièvement les travaux les plus marquants des auteurs ayant œuvré sur l'algorithmique répartie en général et sur la stabilisation en particulier.

Une présentation générale des algorithmes répartis est disponible dans [RH90, Tel94, Lyn96]. Les concepts associés à l'auto-stabilisation sont décrits dans [Sch93, Tel94, Dol00]. Les algorithmes stabilisants proportionnels [1] sont présentés également dans [Dol00]. Les mesures de complexités utilisées par la suite sont détaillées dans [Lav95].

## 1.1 Les systèmes répartis

De nombreux systèmes réels, comme la téléphonie mobile ou les systèmes informatiques, sont organisés en réseaux. Chacun des membres constitue une unité potentiellement autonome pouvant agir et échanger des informations avec un certain nombre d'autres membres. Ce genre de réseau est appelé système réparti et est modélisés par un graphe dont chaque nœud est un processeur muni d'un algorithme. Un exemple de système réparti est donné figure 1.1.

### 1.1.1 Intérêt

Il existe de nombreuses formes de systèmes répartis. Une machine parallèle constituée de plusieurs processeurs partageant la même mémoire est un système réparti (elle peut être modélisée par un graphe complet puisque, via la mémoire partagée, chaque processeur peut échanger de l'information avec tous les autres). Les téléphones (mobiles ou classiques) permettant des échanges verbaux entre différents individus en sont un autre

1. time-adaptif en anglais.

exemple. Plus classiquement, un réseau d'ordinateurs autonomes possédant chacun un processeur et une mémoire propre constitue un système réparti.

Historiquement, les systèmes répartis sont apparus pour faire face à certains problèmes [Tel94] :

1. **Échange d'information :** que ce soit des textes, des images ou des messages auditifs, les échanges d'informations automatisés par l'électronique (par exemple d'ordinateur à ordinateur) sont de loin bien plus rapides que ceux utilisant les moyens classiques de communication. D'où la nécessité d'établir un réseau informatique permettant de tels échanges, ce qui fut fait par les universités et certaines grosses entreprises dans les années soixante.
2. **Partage des ressources :** si le coût des ordinateurs personnels a considérablement baissé, il n'en est pas de même pour un certain nombre de périphériques comme les imprimantes, les scanners ou encore les unités de sauvegarde. Et pour si indispensables qu'ils soient, ces périphériques n'en sont pas moins utilisés qu'occasionnellement. Le développement de réseaux locaux a permis de les rendre accessibles à partir de plusieurs ordinateurs.
3. **Fiabilité :** l'utilisation des systèmes répartis peut permettre un fonctionnement correct du système même en cas de défaillance de certains de ses processeurs. Par exemple, la duplication d'informations sur différentes machines réduit considérablement les risques de perte, tout comme l'exécution de la même application critique sur plusieurs ordinateurs permet de pallier la défaillance d'un certain nombre d'entre eux.
4. **Performance :** le fait de disposer de plusieurs unités de calcul offre la possibilité de diviser une tâche en plusieurs sous tâches et de les faire exécuter simultanément. C'est ce que font les ordinateurs à architectures parallèles. De même, les réseaux permettent en général à un utilisateur d'exécuter des commandes sur plusieurs machines en même temps.

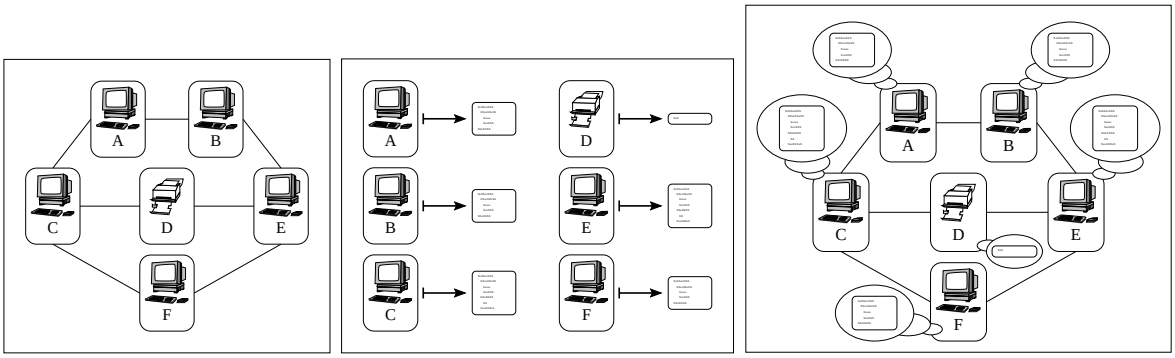
### 1.1.2 Algorithmes répartis

Pour réaliser différentes tâches sur les réseaux tout en exploitant au mieux leurs avantages ont été conçus des algorithmes spécifiques, adaptés au contexte réparti : les algorithmes répartis.

Un algorithme réparti est une fonction associant un protocole à chacun des nœuds d'un réseau (ou graphe). Exécutés simultanément par tous les nœuds, ces protocoles permettent au système de satisfaire certaines propriétés globales. On dit alors que le système est en train de suivre une exécution. Un exemple d'algorithme réparti est donné figure 1.1(b). A chaque nœud du graphe 1.1(a) est associé un protocole. L'ensemble constitue un système réparti (figure 1.1(c)).

D'un point de vue opérationnel, les algorithmes répartis diffèrent des algorithmes centralisés par plusieurs aspects ([Ray85, Tel94]) :

1. **Non-connaissance de l'état global :** dans un système centralisé, le processeur exécutant un protocole connaît l'intégralité du système. Cela demande une grande




(a) Réseau (b) Algorithme réparti (c) Système réparti

FIG. 1.1 – *A quoi rèvent les ordinateurs ?*

capacité de mémoire, puisque toute l'information doit être réunie en un seul nœud. En contrepartie, cela lui permet de faire des choix en utilisant cette connaissance. Dans un système réparti, chaque processeur n'a qu'une connaissance locale de l'état du système. Les processeurs doivent donc faire des choix en disposant d'une information partielle et parfois assez restreinte. S'ils ont besoin d'une information plus complète, ils doivent attendre que cette information leur parvienne. Pour cela, elle doit traverser une partie du réseau, ce qui peut éventuellement la rendre obsolète.

2. **Absence d'horloge globale :** dans un système réparti, chaque processeur possède sa propre horloge. Ces différentes horloges peuvent ne pas être synchronisées. La diversité des multiples composants du réseau peut également induire des différences de vitesse, tout comme l'encombrement des canaux de communication peut rendre aléatoire le temps nécessaire à l'acheminement des messages.
3. **Non-déterminisme :** toutes les différences de vitesse entre les divers constituants du réseau rendent les systèmes répartis non-déterministes. Par exemple, considérons un nœud A devant exécuter son code lorsqu'il reçoit un message d'un voisin B et s'éteindre lorsqu'il reçoit un message d'un autre voisin C. Supposons que B et C lui envoient chacun un message. L'état des canaux reliant A à ses voisins étant arbitraire, rien ne permet de déterminer si A va exécuter son code puis s'éteindre, ou directement s'éteindre. Si le message en provenance de C est particulièrement lent, il est même possible que A reçoive plusieurs messages de B avant de s'éteindre.
4. **Possibilité de résistance aux pannes :** dans le cas d'un algorithme centralisé, si le nœud central est défaillant, tout le système est paralysé. Dans un système réparti, la panne d'un ou plusieurs nœuds peut ne pas être irrémédiable. C'est notamment le cas sur des réseaux de grande taille où les processeurs sont distants les uns des autres. Puisque les actions sont prises en utilisant une information locale, un processeur éloigné de l'endroit où se produit une panne peut continuer à agir.

Cette dernière propriété constitue le cœur des protocoles tolérants aux pannes que nous allons étudier tout au long de cette thèse.

### 1.1.3 Problèmes classiques en algorithmique répartie

Le but des algorithmes répartis est de résoudre au mieux un problème réel. A travers la littérature, parmi tous les problèmes traités, on en trouve un petit nombre qui revient régulièrement. Leur position privilégiée s'explique par deux raisons :

1. **Problèmes cruciaux :** certains de ces problèmes se posent de manière incontournable et critique dans la réalité. Par exemple, il est important que les algorithmes de bas étages des réseaux sur lesquels vont reposer tous les protocoles de communication soient fiables et rapides. D'où une grande nécessité de les étudier.
2. **Comparaison facile :** cette raison n'est que la conséquence directe de la précedente. En effet, la littérature regorge d'articles proposant diverses solutions aux différents problèmes classiques. Les coûts en temps ou en mémoire ont été analysés, des impossibilités ont été établies et certaines bornes d'optimalité ont été atteintes. Ces problèmes "classiques" constituent donc une bonne base de travail permettant de tester l'efficacité d'une nouvelle solution ou d'un nouveau concept. Par exemple, une propriété A peut à première vue sembler identique ou moins intéressante qu'une propriété B ; résoudre selon A un problème classique que la littérature tient pour insoluble selon B établit tout l'intérêt et la spécificité du nouveau concept A.

La liste des problèmes présentés ci-dessous n'est pas exhaustive. Elle se concentre plus particulièrement sur ceux qui ont fait l'objet d'études auto-stabilisantes.

#### A) Problèmes statiques

Les problèmes statiques ([DGS96]) sont ceux dont la spécification ne dépend que de l'état du graphe. C'est généralement le cas lorsque l'on veut que certains processeurs acquièrent des connaissances sur le réseau auquel ils appartiennent (problème des profondeurs), ou encore fassent des choix concertés permettant la réalisation d'un but commum (construction de topologie, bit persistant). En algorithmique réparti classique, une fois que tous ont obtenu les informations recherchées, le problème est résolu et l'exécution peut s'arrêter.

Du point de vue de l'auto-stabilisation, un système peut être sujet à des corruptions. Elles peuvent naturellement intervenir après que le système ait trouvé une solution au problème. D'où, lorsqu'un algorithme stabilisant résoud un problème statique, il continue à effectuer un certain nombre de contrôles même après avoir trouvé une solution. Néanmoins, lorsque les seules actions possibles des processeurs consistent à mettre à jour leurs variables en leur donnant une valeur identique à celle qu'elles avaient déjà, on considére que l'exécution est terminée.

**Construction de topologie** Ce problème se pose lorsqu'on veut adapter un algorithme à un réseau pour lequel il n'a pas été conçu. Par exemple, l'exclusion mutuelle connaît de nombreuses solutions efficaces sur un anneau mais assez peu sur un graphe quelconque. Construire une topologie d'anneau sur un graphe permet, en composant les deux algorithmes, d'obtenir un algorithme d'exécution mutuelle sur un graphe.

Chen, Yu et Huang proposent une construction d'arbre auto-stabilisante à partir d'un graphe quelconque dans [CYH91]. Une consturction [BLB95]

**Diffusion & Échange total** Les systèmes répartis ne disposent à priori que d'un mode de communication point à point ; lorsqu'un nœud veut faire circuler une information, il ne peut la faire parvenir qu'aux processeurs ayant un lien de communication direct avec lui. Pour certains problèmes, il peut être nécessaire d'utiliser des primitives de communication globale. La diffusion consiste pour un processeur à faire parvenir son information à tous les autres.

L'échange total est une généralisation de la diffusion. Il ne s'agit plus de faire communiquer un processeur avec le reste du réseau mais tous les processeurs doivent communiquer avec tous les autres.

Les problèmes de diffusion et d'échange total sont traités dans [JdR94] dans un cadre non auto-stabilisant. Un algorithme de diffusion parallèle auto-stabilisant est proposé dans [JADT99].

**Élection** Même dans le cadre réparti, il peut arriver que certaines applications aient besoin momentanément de distinguer un processeur, que ce soit pour lui faire exécuter un code différent ou pour prendre des décisions globales concernant l'ensemble du réseau. L'élection consiste à faire désigner un unique processeur par l'ensemble du réseau.

Lee Lann donne un algorithme d'élection probabiliste sur un anneau dans [GlL77]. Afek et Gafni présentent une étude sur la complexité de l'élection dans [AG91].

**Profondeur** Ce problème se pose sur les graphes semi-uniformes. Le processeur étiqueté différemment des autres est appelé le processeur "distingué", ou la racine du graphe. C'est fréquemment le cas dans les réseaux de type "arbre" ou "en étoile". Ces topologies permettent en général d'allier les avantages des systèmes répartis et la puissance de décision des systèmes centralisés. Dans un tel contexte, il peut devenir intéressant pour un processeur quelconque de connaître le nombre de processeurs le séparant de la racine du graphe. Ce nombre est appelé profondeur du processeur, ou distance à la racine. Résoudre le problème des profondeurs consiste à faire connaître à chaque processeur sa distance à la racine.

Ce problème est souvent traité de la même manière qu'une construction d'arbre sur un réseau ou qu'un routage des plus courts chemins [SG89, Her91].

**Bit persistant** Ce problème est en apparence un des plus simples qui soient : tous les processeurs disposent d'une variable qui doit être identique sur tout le réseau. Dans la pratique, cela permet de gérer la duplication d'informations ou de vérifier la stabilité d'un système.

Ce problème est présenté dans [KP95]. Une version auto-stabilisante et $k$-stabilisante proportionnelle est donnée dans [KP97]

### B) Problèmes dynamiques

A l'inverse, les problèmes dynamiques ne dépendent pas uniquement de l'état du réseau mais plus de son évolution à travers le temps. En effet, les spécifications de ces problèmes portent sur les exécutions du système réparti. C'est notamment le cas lorsqu'on veut qu'une information circule en permanence entre tous les processeurs (exclusion mutuelle) ou que tous disposent de la même information régulièrement mise à jour (synchronisation). En algorithmique stabilisante comme en algorithmique classique, les exécutions liées à ce genre de problème sont toujours infinies.

**Exclusion mutuelle** Un système réparti vérifie la spécification "exclusion mutuelle" si à tout moment, un seul de ses processeurs a accès à une certaine ressource et si un processeur désirant la ressource est certain d'y avoir accès. Ce genre de problèmes se pose notamment lors de la mise en commun des moyens de production, comme le partage d'une imprimante par plusieurs ordinateurs. Deux ordinateurs ne doivent pas pouvoir imprimer simultanément et tous doivent être sûrs qu'il auront tout de même accès au service ultérieurement. Les références sur l'exclusion mutuelle sont présentées section 0.3.3, page 8

**k exclusion mutuelle** C'est une variante de l'exclusion mutuelle. A chaque instant, exactement $k$ ressources doivent être présentes dans le système. Là encore, chaque processeur doit être assuré d'avoir accès à chacune d'entre elles infiniment souvent.

Ce problème est défini dans [FLBB82]. Des versions auto-stabilisantes sous différentes hypothèses peuvent être trouvées dans [FDS94, ADHK97]

**Synchronisation** Les systèmes répartis ne disposent pas d'horloge globale (paragraphe 2, page 13). La nature des différents composants induit des variations de vitesses et entraîne un non-déterminisme. Pour certains problèmes, cela peut devenir gênant. La synchronisation d'un réseau consiste à maintenir artificiellement une horloge dans chaque processeur et à gérer toutes les horloges de manière à ce qu'elles indiquent le même temps au même moment.

La synchronisation a été introduite par B. Awerbuch dans [Awe85].

## 1.2 Hypothèses de modélisation

De par leur grande diversité, les réseaux présentent des comportements très variables aussi bien au point de vue du mode de communication que du fonctionnement interne des processeurs. Néanmoins, il est possible de les classifier en groupes selon certaines caractéristiques communes, caractéristiques qui servent ensuite d'hypothèse aux algorithmes répartis.

### 1.2.1 Système de communication

En premier lieu viennent les hypothèses de communication : un réseau est constitué de divers éléments pouvant communiquer les uns avec les autres. Cette communication peut être modélisée de différentes manières :

1. **Echange de message :** lorsqu'un processeur veut donner une information à un autre, il lui envoie un message via un canal de communication. Ce modèle prend en compte le temps qui s'écoule entre l'expédition d'un message et sa réception (délai de transmission) ainsi que les possibilités d'apparition, de disparition ou de duplication de message. Il considère les liens de communication comme des entités à part entière ayant des caractéristiques propres, notamment ([Lyn96]) :
   (a) **Canaux à mémoire bornée / non bornée :** un canal peut accepter simultanément un nombre variable de messages. Certain canaux n'en tolèrent qu'un nombre fini fixé à l'avance. Ils sont appelés canaux à mémoire bornée. Les autres, à mémoire non bornée, en acceptent un nombre illimité.
   (b) **FIFO / non FIFO :** FIFO vient de l'anglais First In First Out (Premier Entré, Premier Sorti). Les canaux ayant cette propriété assurent que l'ordre de réception sera le même que celui de l'émission. Les canaux non FIFO autorisent un message à en dépasser un autre.
   (c) **Communication à délai borné / non borné :** des canaux à délai borné assurent que la durée de transmission des messages ne dépasse pas un certain temps. Les autres, à délai non borné, ne donnent aucune garantie sur le délai d'acheminement.
   (d) **Unidirectionnel / bidirectionnel :** un canal reliant deux processeurs $P$ et $Q$ peut être capable d'acheminer des messages de $P$ vers $Q$ et de $Q$ vers $P$, ou n'être capable que de l'une de ces deux opérations. Dans le premier cas, le canal est bidirectionnel, dans le second il est unidirectionnel.
2. **Mémoire partagée :** Dans le modèle à mémoire partagée, les processeurs disposent d'une zone mémoire commune dans laquelle ils peuvent écrire et lire des informations. Là encore, ces caractéristiques propres sont variables :
   (a) **Mémoires spécifiques / indifférenciées :** quand plusieurs processeurs partagent une même mémoire, certains peuvent se voir imposer des restrictions quant aux opérations qu'ils sont habilités à effectuer. Classiquement, un processeur peut être autorisé à lire le contenu d'une mémoire, mais pas à y écrire, ou inversement.
   (b) **Type de registres :** si un processeur lit un registre pendant que d'autres processeurs y écrivent, plusieurs choses peuvent se produire, selon la fiabilité du registre :
      i. **Registre sûr :** la lecture renvoie une des valeurs quelconques possibles pour le registre.
      ii. **Registre régulier :** les registres réguliers assurent qu'une lecture retournera soit la valeur du registre avant écriture, soit une des valeurs en train d'être écrite.

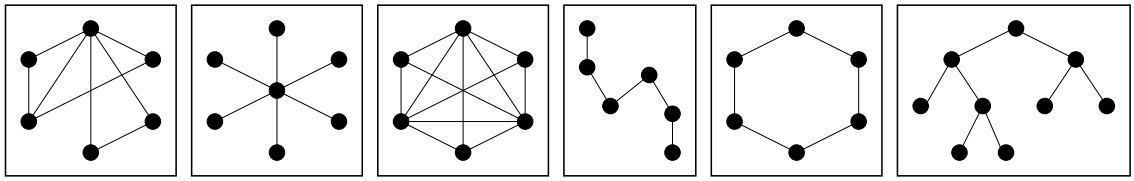



FIG. 1.2 – *Quelques exemples de graphe*

iii. **Registre atomique :** les opérations de lecture / écriture sont considérées comme atomiques. La lecture doit intervenir avant ou après l'écriture, mais pas simultanément.

(c) **Unidirectionnel / bidirectionnel :** un processeur $P$ peut avoir un accès en lecture au registre d'un processeur $Q$ sans que $Q$ ait un accès à une des mémoires de $P$. Dans ce cas, $P$ peut disposer d'informations sur $Q$ sans que l'inverse soit vrai. Ce genre de réseau est unidirectionnel. Dans un réseau bidirectionnel, pouvoir lire la variable d'un autre processeur implique une réciproque.

Le modèle à mémoire partagée peut sembler moins pertinent que celui à passage de message. En effet, la mémoire partagée peut-être considérée comme un canal acceptant un unique message (qui est donc perdu dès qu'un nouveau message est envoyé) réémis à chaque fois qu'il arrive. Mais il présente de grands avantages au point de vue formalisation des algorithmes et de la simplicité des preuves. De plus, de nombreuses études montrent que, sous certaines hypothèses, les deux modéles ont un même pouvoir d'expressivité et proposent des méthodes pour adapter les algorithmes de l'un à l'autre [AB93, KP93, Dol00].

Dans la suite de cette thèse, nous n'utiliserons que le modèle à mémoire partagée.

### 1.2.2 Topologie

La topologie du réseau a elle aussi une importance déterminante, notamment par rapport aux opérations autorisées aux algorithmes. Par exemple, si tous les processeurs disposent deux à deux d'un lien de communication, les problèmes liés à l'échange d'information sont grandement simplifiés. La topologie d'un réseau dépend de deux paramètres :

1. **Structure du réseau :** les processeurs sont reliés entre eux par des liens de communication. Ces liens définissent la structure du réseau. Certaines d'entre elles offrent des propriétés particuliéres ([JdR94]).
   (a) **Réseau complet :** chaque processeur dispose d'un lien de communication avec tous les autres processeurs (Figure1.2.(c)).
   (b) **Etoile :** un processeur a un rôle central ; il est relié à tous les autres processeurs. Par ailleurs, ces derniers ne sont pas reliés entre eux (Figure1.2.(b)).

(c) **Chaîne :** une chaîne est un réseau ordonnable de manière à ce que chaque processeur (sauf le plus petit et le plus grand) soit relié uniquement à son successeur et son prédécesseur immédiat (Figure1.2.(d)).

(d) **Anneau :** un anneau est une chaîne à laquelle un lien de communication entre le plus grand et le plus petit des processeurs est ajouté (Figure1.2.(e)).

(e) **Arbre :** un réseau a une structure d'arbre s'il existe une numérotation des processeurs tel que chaque processeur dispose d'au plus un lien le reliant à un processeur ayant un numéro plus petit que lui (Figure1.2.(f)).

2. **Uniformité :** certain réseaux autorisent un ou plusieurs processeurs à avoir un rôle différent. Plus précisement, on distingue :

(a) **Réseau non-uniforme :** tous les processeurs sont susceptibles d'être distingués les uns des autres, par exemple en utilisant des identificateurs.

(b) **Réseau semi-uniforme :** un seul processeur est distingué. Les autres ne sont pas identifiables.

(c) **Réseau anonyme :** tous les processeurs sont identiques. Cela entraîne entre autre chose que tous les processeurs exécutent le même code.

### 1.2.3 Démon

La dernière hypothèse de modélisation concerne le démon.

Dans un système réparti, chaque nœud dispose d'un protocole lui indiquant les actions qu'il doit accomplir. Nous l'avons déjà mentionné, les différences de vitesse entre les composants du réseau entraînent un non-deterministe. A priori, rien ne permet de savoir comment une exécution va se dérouler.

Du point de vue de la modélisation, une telle multitude d'exécution rend la conception et la démonstration d'algorithme particulièrement difficile à établir. D'où l'introduction du *démon*.

Un démon est une hypothèse simplificatrice faite sur l'ensemble des exécutions d'un système réparti. Classiquement, il limite le nombre des exécutions possibles d'un système, rendant ainsi possible la formalisation et la preuve d'algorithmes. Il se présente sous forme d'un prédicat sur les exécutions du système. Une exécution ne vérifiant pas le prédicat est considérée comme ne pouvant pas se produire et n'a pas à être considérée dans la preuve de l'algorithme.

Concrètement, la majorité des démons fixe un ensemble de contraintes que les exécutions des systèmes répartis doivent suivre. En premier lieu, ils fixent l'atomicité des actions exécutables par les processeurs ([Tix00]). Une action atomique est une action non interruptible par une autre action. En particulier, rien ne peut survenir entre le début et la fin de son exécution. Ensuite, le démon détermine si plusieurs processeurs peuvent agir simultanément ou si les actions seront considérées comme étant exécutées les unes après les autres. Enfin, il définit l'ordre dans lequel les processeurs auront le droit d'agir.

**Anthropomorphisme démoniaque :** Formellement, le démon est un objet statique qui fixe les principes régissant les exécutions. L'ensemble des exécutions d'un système est

alors représentable par une forêt dont certaines branches sont interdites par le démon.

Dans la pratique, lorsque l'on cherche à prouver une propriété du système, on doit prendre en compte toutes les exécutions possibles. Sur certaines d'entre elles la preuve peut être facile, sur d'autres elle peut être délicate, mais toutes doivent être considérées. Aussi, le démon est-il souvent considéré comme un adversaire ayant un pouvoir : celui de choisir l'exécution la moins favorable à la propriété que l'on cherche à établir. Par exemple, si la spécification du démon accepte les exécutions dans lesquelles certains processeurs n'agissent pas alors que d'autres agissent plusieurs fois, on parlera d'un démon qui a le pouvoir de laisser certains processeurs inactifs tout en en favorisant d'autres.

**Démons classiques :** La littérature définit un grand nombre de démons ([DIM93, Dij74, TH94, IJ93, DT00, Tix00]). Nous nous limiterons ici à l'étude de ceux utilisés dans le modèle à état. Classiquement, le modèle à état considère principalement cinq types de démons, chacun ayant une définition propre de l'atomicité.

1. **Démon lecture / écriture ([DIM93]) :** ce démon considère que seules les actions de lecture ou d'écriture sont atomiques. A part cela, aucun ordre n'est imposé quant à l'exécution des actions. En particulier, un processeur $P$ ayant deux voisins $Q_1$ et $Q_2$ peut lire une variable de son voisin $Q_1$. $Q_1$ et $Q_2$ peuvent ensuite être amenés à modifier leurs valeurs. Ensuite, $P$ peut être autorisé à lire la variable de $Q_2$. Lorsque $P$ agira, il le fera en fonction des valeurs supposées de $Q_1$ et $Q_2$, valeurs que ses voisins n'auront peut-être jamais eues simultanément.
2. **Démon totalement réparti ([TH94]) :** ce démon autorise un processeur à lire l'ensemble des variables de ses voisins ou à écrire dans toutes ses variables propres en une action. A part cela, aucun ordre n'est imposé. En particulier, un processeur $P$ peut lire les variables d'un voisin $Q$, suite à quoi $Q$ peut effectuer une modification de ses valeurs qui sera suivie par l'action de $P$. Les valeurs lues par $P$ sont alors obsolètes. Mais le Démon totalement réparti autorise tout de même $P$ à agir en fonction des anciennes valeurs de $Q$.
3. **Démon réparti ([Dij74]) :** le démon choisit un ensemble de processeurs $\mathcal{P}$ parmi ceux qui peuvent agir. Puis tous les processeurs de $\mathcal{P}$ doivent simultanément lire les variables de leurs voisins. Enfin, ils doivent tous agir. L'ensemble constitue une action atomique.
4. **Démon centralisé ([Dij74]) :** ce démon agit comme un démon réparti mis à part que l'ensemble des processeurs activables $\mathcal{P}$ ne doit contenir qu'un seul élément. Ensuite, cet unique processeur doit lire les variables de ses voisins, puis agir.
5. **Démon synchrone ([Dij74, Her90]):** ce démon agit comme un démon réparti activant à chaque fois le plus de processeurs possibles. $\mathcal{P}$ doit contenir tous les processeurs pouvant agir. Ils doivent tous simultanément lire les variables de leur voisin, puis agir ensemble.

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

1. **Démon non-équitable :** les démons non équitables sont totalement libres. En particulier, si un processeur peut continuellement agir, le démon peut le choisir

éternellement, privant ainsi d'autres processeurs de l'opportunité d'exécuter leurs actions.

2. **Démon équitable :** les démons équitables interdisent qu'un processeur pouvant agir soit systématiquement exclu de $\mathcal{P}$.
3. **Démon probabiliste :** ces démons choisissent l'ensemble $\mathcal{P}$ aléatoirement, généralement en fonction d'une loi de probabilité.

Par la suite, nous ne considèrerons que des démons non-équitables centralisés ou synchrones (les démons synchrones sont nécessairement équitables).

## 1.3 Tolérance aux défaillancess

Une des propriétés des systèmes répartis est de pouvoir résister aux défaillances. Celles que nous considérons sont principalement de deux types :

1. **Pannes définitives :** ces pannes sont en général dues à une panne matérielle ou à une erreur dans le code du programme. Selon leur nature et le moment où elles se déclenchent, elles se divisent en deux catégories :
   (a) **Panne totale :** un processus sujet à une panne totale ne peut plus recevoir ou envoyer de messages. Ce genre de panne correspond généralement à une défaillance matérielle franche.
   (b) **Défaillance byzantine :** le comportement d'un processus byzantin est quelconque. En particulier, il peut ne pas suivre les spécifications de son algorithme. Ce type de panne est le plus souvent lié à une erreur de programmation ou une défaillance matérielle chronique.
2. **Défaillances transitoires :** rien de définitif dans les pannes transitoires puisque seules les mémoires du système sont atteintes. Le code du programme et les capacités de communication des processeurs ne sont pas affectés. Là encore, on distingue deux types de pannes :
   (a) **Corruption totale :** tout le réseau est affecté. Cela signifie que le contenu des mémoires, mais aussi celui des canaux de communication peut être quelconque.
   (b) **Corruption partielle :** seul un nombre fixé de processeurs ou de canaux sont sujets à une corruption. Les autres conservent la valeur qu'ils avaient au moment où la panne frappe le réseau.

Pour des raisons historiques, le domaine traitant des pannes définitives s'appelle "tolérance aux pannes" alors que les pannes transitoires sont du ressort de l'auto-stabilisation.

### 1.3.1 Auto-stabilisation

L'auto-stabilisation est une propriété des systèmes répartis qui leur assure de retrouver un comportement correct après une corruption de leurs mémoires. Après la panne, l'algorithme traverse une phase transitoire (appelée phase de stabilisation) au cours de laquelle les spécifications du système peuvent ne plus être satisfaites. Mais inévitablement cette phase conduit le système vers une exécution correcte. Une illustration du

fonctionnement des algorithmes auto-stabilisants est donnée figure 1.3. En partant d'un état du système quelconque (état 1), le système passe par un certain nombre d'états (phase de stabilisation, états 2 à 18) puis retrouve un comportement vérifiant la spécification du problème (phase stabilisée, états 19 à 27 et suivant).

L'auto-stabilisation tolère les pannes sous le couvert de deux hypothèses :

1. **Code fiable :** les pannes peuvent affecter les composants volatiles du système, mais pas la mémoire morte. Cela signifie que toutes les mémoires vives peuvent être corrompues, ainsi que tous les liens de communication. Par contre, cela exclut le code du programme lui-même.
2. **Faible fréquence des pannes :** les pannes frappant le système doivent être suffisamment éloignées dans le temps les unes des autres pour que l'algorithme puisse retrouver un comportement correct entre deux pannes. Après une corruption mémoire, un algorithme auto-stabilisant met un certain temps avant de ramener le système dans un état correct. Si des pannes interviennent trop régulièrement, la propriété de retour vers une exécution vérifiant la spécification du problème peut ne plus être assurée.

Aucune hypothèse n'est faite sur le nombre de processeurs affectés par une corruption mémorielle, pas plus que sur la nature des pertes ou apparitions de message dans les liens de communication.

Outre la résistance aux pannes transitoires, les algorithmes auto-stabilisants offrent plusieurs avantages :

1. **Absence d'initialisation :** un système réparti dont aucun des processeurs ne serait initialisé peut être considéré comme un réseau qui vient d'être totalement corrompu. Dans ce cas, un algorithme auto-stabilisant assure tout de même un retour vers un fonctionnement correct. L'initialisation du réseau, parfois si contraignante, n'est donc pas nécessaire pour faire fonctionner un algorithme auto-stabilisant.
2. **Réseau dynamique :** certains systèmes répartis évoluent au cours du temps, par exemple un groupe de processeurs peut être ajouté au reste du réseau. Si les hypothèses de non-corruption du code (c'est à dire si le code est présent dans la mémoire morte des processeurs ajoutés) et de faible fréquence entre les pannes (les modifications de topologie sont ici considérées comme des pannes) sont respectées, alors l'algorithme assure un retour vers un fonctionnement correct.

L'utilisation des algorithmes auto-stabilisants présente également un certain nombre d'inconvénients :

1. **Non-détection de la terminaison :** chaque processeur n'a qu'une vision locale de l'état du système. Les processeurs peuvent ne pas être capables de détecter les pannes, ou encore une panne peut affecter une partie seulement du réseau. Dans ces deux cas, certains processeurs vont avoir le même comportement que lorsque le système fonctionne bien. Il leur est impossible de faire la différence entre ce genre de situation et une exécution stabilisée. De manière générale, lorsque le problème traité est global, aucun processeur n'est à même de détecter la fin de la phase de stabilisation.

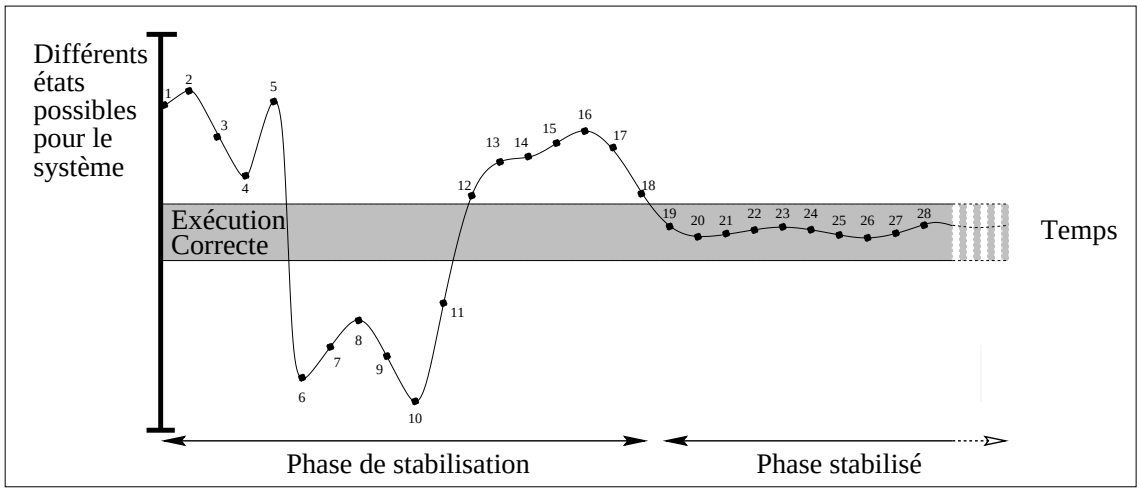



FIG. 1.3 – *Phases de l'auto-stabilisation*

2. **Surcoût de fonctionnement :** en permanence, les processeurs exécutent leur code. Ce code contient des mécanismes permettant un retour vers un fonctionnement correct après perturbation. Lorsqu'il n'y a pas de panne, ces mécanismes sont tout de même activés ; en effet, comme les processeurs ne peuvent détecter le retour à la normale, ils doivent les exécuter en permanence. Par rapport à un algorithme réparti non auto-stabilisant présentant les mêmes services en cas de bon fonctionnement, cela entraîne généralement un surcoût en mémoire ou en nombre de messages échangés entre les processeurs.
3. **Absence de contrôle durant la phase de stabilisation :** après une corruption des mémoires, l'auto-stabilisation assure un retour vers un comportement correct. Mais aucune propriété n'est assurée pendant la phase de stabilisation. En particulier, une grande partie du réseau peut se considérer comme correcte et agir en conséquence alors qu'il n'en est rien.
4. **Longue phase de stabilisation :** à cet égard, l'auto-stabilisation peut être comparée à un service après-vente. Le concept est intéressant si les délais de réparation sont raisonnables. Malheureusement, la réalité est autre et il arrive fréquemment qu'une simple panne puisse entraîner une longue phase de stabilisation.

Les algorithmes auto-stabilisants tolèrent des pannes pouvant affecter la totalité d'un système réparti. Mais en cas de panne légère, ils utilisent les mêmes mécanismes de correction qu'en cas de panne totale, et la valeur corrompue d'un simple processeur peut parfois se propager et corrompre tout le reste du réseau, rendant ainsi la phase de stabilisation relativement longue. On serait en droit d'espérer un traitement plus local et plus rapide des fautes singulières, ainsi que leur circonscription à une zone proche des processeurs corrompus. Les algorithmes $k$-stabilisants présentent des solutions allant dans ce sens.

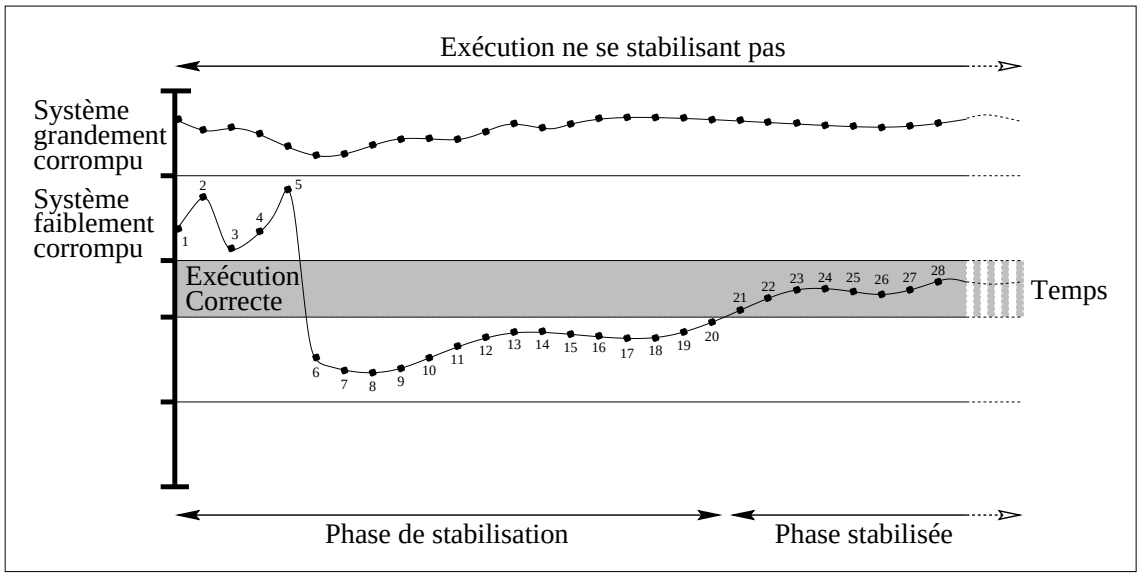



FIG. 1.4 – *Principes de la k-stabilisation*

### 1.3.2 $K$-stabilisation

La $k$-stabilisation est une généralisation de l'auto-stabilisation. C'est une propriété des algorithmes répartis qui assure à un système de retrouver un comportement correct après une corruption des mémoires. Mais contrairement à l'auto-stabilisation, la $k$-stabilisation fait des hypothèses sur le nombre de processeurs du réseau pouvant être affectés par la panne.

La $k$-stabilisation tolère les pannes sous le couvert de trois hypothèses (les deux premières étant les mêmes que celles de l'auto-stabilisation) :

1. **Code fiable :** les mémoires vives et les liens du système peuvent être corrompus, mais pas le code du programme.
2. **Faible fréquence des pannes :** la fréquence entre les pannes doit être suffisamment faible pour permettre à l'algorithme de retrouver un comportement correct entre deux pannes.
3. **Pannes de taille variable :** l'auto-stabilisation autorise la corruption de l'intégralité du réseau. La $k$-stabilisation considére des pannes pouvant affecter tout le graphe ou une partie du graphe seulement. Plus précisément, un algorithme $k$-stabilisant est conçu pour résister à des pannes affectant au plus $k$ processeurs. Dans le cas où $k$ est la taille du graphe, les concepts de $k$-stabilisation et d'auto-stabilisation sont identiques. Dans le cas inverse, un système $k$-stabilisant frappé par une panne affectant plus de $k$ processeur n'est pas sûr de se stabiliser.

Dans le cas de pannes non totales, la dernière hypothèse entraîne quelques affaiblissement des avantages que présentaient les algorithmes auto-stabilisants. Par exemple la $k$-stabilisation nécessite une initialisation de tous les processeurs sauf d'au plus $k$ d'entre eux. De même, dans le cas de réseaux dynamiques, au plus $k$ processeurs peuvent être ajoutés. Ensuite, il faut attendre la stabilisation du système pour pouvoir à nouveau

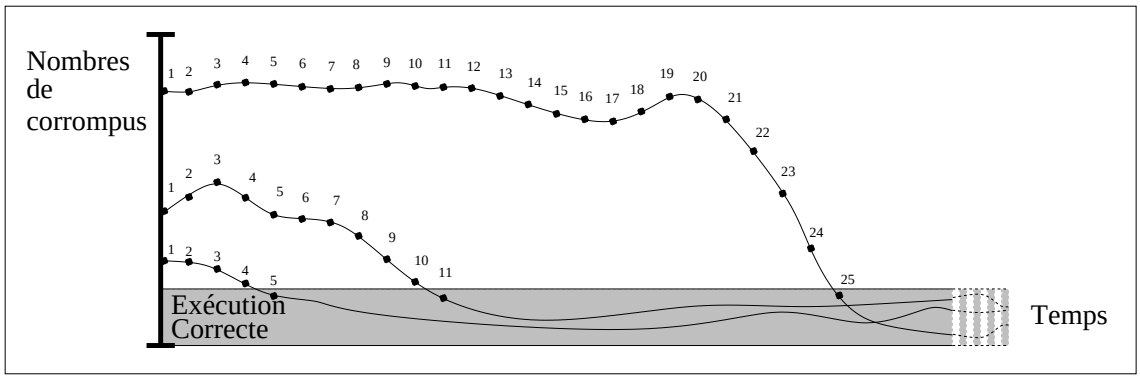



FIG. 1.5 – *Trois exemples d'exécutions stabilisante proportionnellement*

faire une modification du réseau.

Une illustration du fonctionnement des algorithmes $k$-stabilisants est donnée figure 1.4. A partir d'un état du système faiblement corrompu (état 1), l'exécution se stabilise. Par contre, à partir d'un état fortement corrompu, le système n'est pas assuré de retrouver un état correct.

### 1.3.3 Stabilisation proportionnelle

Les algorithmes $k$-stabilisants ne présentent aucune garantie sur le temps de stabilisation. En particulier, même un algorithme ne tolérant qu'un petit nombre de fautes peut converger lentement. Les algorithmes stabilisants proportionnellement aux nombres de fautes (ou algorithme proportionnel) remédient à cet inconvénient. Un algorithme réparti est proportionnel si sa phase de stabilisation est proportionnelle au nombre effectif de fautes frappant le réseau. Tout comme l'auto-stabilisation, la stabilisation proportionnelle tolère les corruptions de mémoire vive, mais pas celle du code du programme.

A priori, les algorithmes proportionnels ne présentent que des avantages :

1. **Temps de stabilisation proportionnel aux fautes :** la stabilisation proportionnelle assure un retour vers un état normal en un temps proportionnel au nombre effectif de fautes. Cela signifie que la phase de stabilisation sera très rapide dans le cas d'une unique faute tout en restant raisonnable dans le cas d'une corruption d'une petite partie du réseau. Cela garantit également toutes les propriétés propres à l'auto-stabilisation. La phase de stabilisation sera peut-être longue (proportionnelle à la taille du réseau), mais le système convergera vers un fonctionnement correct, comme c'est le cas pour l'auto-stabilisation.
2. **Propagation proportionnelle aux fautes :** la taille de la zone de propagation des erreurs est bornée. En effet, les transmissions de messages entre processeurs prennent du temps. La phase de stabilisation étant courte, un processeur physiquement éloigné d'un lieu de corruption n'aura donc pas le temps de recevoir un message en provenance d'un processeur corrompu. En cas de faible corruption, les erreurs ne peuvent pas se propager loin du lieu de défaillance.

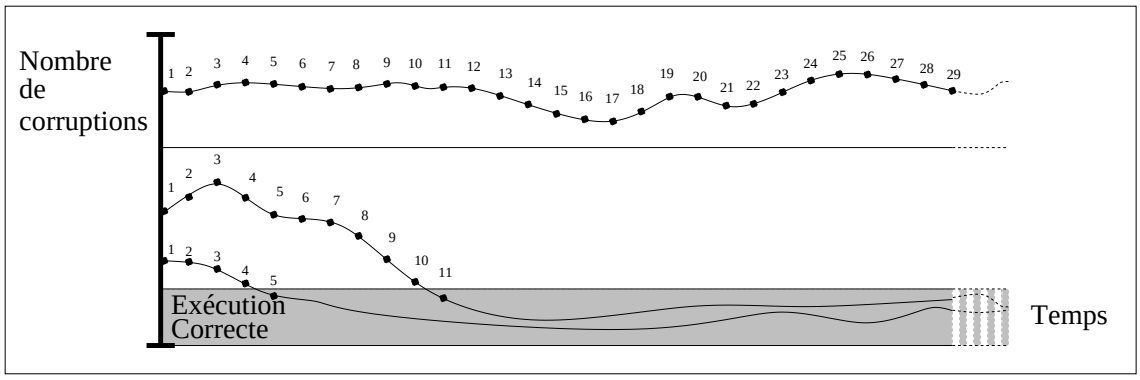



FIG. 1.6 – *Trois exemples d'exécutions k-proportionnelles*

Une illustration du fonctionnement des algorithmes proportionnels est donné figure 1.5. La première exécution représentée part d'un état très faiblement corrompu et le système retrouve assez vite un comportement correct alors que la troisième exécution a une phase de stabilisation très longue.

### 1.3.4 Stabilisation $k$-proportionnelle

Dans la littérature, les algorithmes proportionnels sont rares, sans doute parce que difficiles à découvrir. Aussi, un certain nombre d'auteurs préfèrent-ils ajouter une hypothèse sur le nombre de corruptions autorisées. Un algorithme réparti est $k$-proportionnel si sa phase de stabilisation est proportionnelle au nombre effectif de fautes, dans la mesure où au plus $k$ fautes frappent le réseau. En cas de panne plus large, la stabilisation $k$-proportionnelle n'assure plus de retour à la normale. Là encore, lorsque $k$ est la taille du réseau, les concepts de stabilisation proportionnelle et de stabilisation $k$-proportionnelle se rejoignent.

Les algorithmes $k$-proportionnels présentent les mêmes avantages que les algorithmes proportionnels :

1. **Temps de stabilisation proportionnel aux fautes :** le retour vers la normale se fait au prorata du nombre de fautes.
2. **Propagation proportionnelle aux fautes :** les erreurs ne peuvent pas se propager loin du lieu de défaillance.

Ils présentent également les inconvénients de la $k$-stabilisation (nécessité d'un **code fiable**, **faible fréquence des pannes** et **pannes de taille limitée**.

### 1.3.5 Algorithmes anti-corruption

Même si la stabilisation proportionnelle limite la propagation des erreurs, elle ne peut totalement éviter la corruption d'une partie plus ou moins grande du réseau. Classiquement, un processeur corrompu voisin d'un processeur correct peut lui donner de fausses informations et ainsi laisser de fausses valeurs se propager au reste du réseau. Éviter cela est l'objectif principal de la stabilisation anti-corruption. Le principe de base

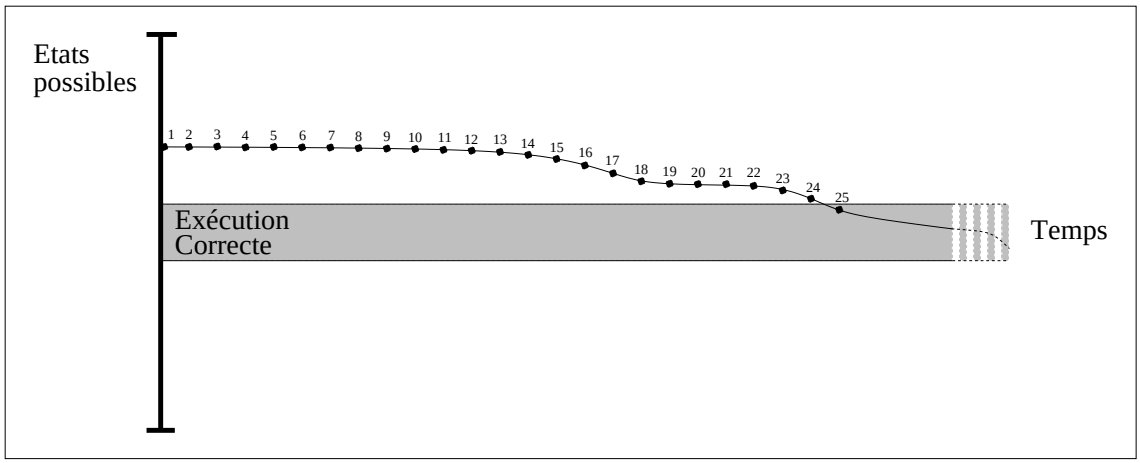



FIG. 1.7 – *Exécution anti-corruption*

est simple : ce qui est sain doit le rester. Un algorithme anti-corruption est donc un algorithme réparti assurant aux processeurs non directement touchés par une panne externe de garder des valeurs correctes jusqu'à la stabilisation du système.

Les avantages des algorithmes anti-corruption tiennent principalement en leur grande fiabilité. Par exemple, dans un système multi-utilisateurs, les divers composants du réseau peuvent présenter des risques de défaillance plus ou moins importants. Un ordinateur récent et correctement configuré peut être plus fiable qu'une vieille machine mal entretenue n'offrant plus qu'une utilisation épisodique. Dans ce genre de cas, les algorithmes anti-corruption offrent la certitude que les diverses pannes des chaînons faibles de réseau ne viendront pas perturber ses autres composants.

Une illustration du fonctionnement des algorithmes anti-corruption est donnée figure 1.7. L'exécution représentée part d'un état très faiblement corrompu. Le nombre de corrompus n'augmente jamais.

### 1.3.6 Thèmes et variations

Dans la pratique, la classification formelle présentée ci-dessus connaît de nombreuses variations. Idéalement, un algorithme à la fois proportionnel et anti-corruption semble optimal : phase de stabilisation courte en cas de corruption de petite taille, non propagation des erreurs tout en assurant l'auto-stabilisation en cas de corruption totale. Mais de tels algorithmes sont particulièrement difficiles à concevoir. Aussi la littérature présente-t-elle de nombreuses variantes.

1. $k$ **anti-corruption :** le côté statique des algorithmes anti-corruption peut compromettre leur côté auto-stabilisant. Ainsi, si une grosse majorité du réseau est corrompue, il peut être intéressant de considérer les corruptions comme les bonnes valeurs et les anciennes bonnes valeurs comme des corruptions. L'inertie des algorithmes anti-corruption ne permet pas une telle approche. C'est pourquoi, on peut leur ajouter la même restriction que pour les algorithmes auto-stabilisants et

proportionnels, à savoir une tolérance à un petit nombre de pannes. Là encore, la propriété d'auto-stabilisation est potentiellement perdue.

2. **Stabilisation $k$-proportionnelles et auto-stabilisante :** à l'inverse, les algorithmes $k$-proportionnels n'incluent pas de clause auto-stabilisante, mais il est toujours possible d'en ajouter une. On obtient ainsi un algorithme ayant une phase de stabilisation proportionnelle au nombre de fautes lorsqu'un petit nombre de pannes survient tout en garantissant la convergence quand le nombre de corruptions dépasse $k$.

### 1.3.7 Relation entre les diverses classes d'algorithmes stabilisants

| | | Phase de stabilisation Proportionnelle aux fautes ↓ | | Phase de stabilisation quelconque ↓ |
|---|---|---|---|---|
| Pannes affectant tout le réseau | $\longrightarrow$ | Stabilisation proportionnelle | $\subset$ | Auto-Stabilisation |
| | | $\cap$ | | $\cap$ |
| Pannes affectant au plus $k$ procésseurs | $\longrightarrow$ | Stabilisation $k$-proportionnelle | $\subset$ | $k$-Stabilisation |

FIG. 1.8 – *Relation entre les différents concepts de stabilisation*

D'un point de vue purement formel, la stabilisation proportionnelle forme une sous-classe de l'auto-stabilisation. En effet, avant d'avoir un temps de stabilisation court, un algorithme proportionnel satisfait aux propriétés de convergence et de correction. Il est donc auto-stabilisant. De la même manière, la stabilisation $k$-proportionnelle est une sous classe de la $k$-stabilisation.

Parallèlement à cela, l'auto-stabilisation peut être considérée comme un cas particulier de la $k$-stabilisation, cas où le nombre de fautes toléré est aussi grand que la taille du réseau. De même, la stabilisation proportionnelle est un cas particulier de la stabilisation $k$-proportionnelle.

Les liens entre les différentes classes d'algorithmes stabilisants sont résumés figure 1.8.

### 1.3.8 Critères d'efficacité

Lorsque plusieurs solutions à un même problème sont proposées, il est intéressant de pouvoir évaluer leurs performances respectives. On peut ainsi déterminer celle qui à priori serait la plus efficace en cas d'implantation. Différents critères permettent cette mesure.

1. **Complexité en espace :** au cours de l'exécution d'un algorithme, la majorité des processeurs doivent stocker un certain nombre d'informations. Pour cela, ils ont

besoin de mémoire. Naturellement, plus un algorithme nécessite de mémoire, plus sa mise en œuvre sera coûteuse. La complexité en espace est la taille des variables (mesurée en bits, en octets en ou nombre d'états) nécessaires à chaque processeur pour assurer la bonne marche de l'algorithme. Dans le cas où tous les processeurs n'ont pas les mêmes besoins, on peut considérer deux différentes méthodes de mesure.

   (a) **Complexité au pire :** on considère la taille de la mémoire utilisée par le processeur le plus gourmand.
   (b) **Complexité totale :** on somme la taille des mémoires nécessaires à chaque processeur.

2. **Complexité en temps :** les algorithmes auto-stabilisants, $k$-stabilisant, proportionnel et anti-corruption présentent tous une phase de stabilisation. Nous l'avons déjà précisé (paragraphe 3, page 23), aucune garantie n'est donnée quant au fonctionnement correct de l'algorithme durant cette phase. Aussi il est important de la réduire le plus possible. La complexité en temps est une mesure évaluant le nombre d'étapes séparant le début d'une exécution de son retour à un fonctionnement correct. Là encore, on distingue plusieurs types de complexité :
   (a) **Complexité au pire :** cette mesure est la plus usitée. L'exécution d'un algorithme stabilisant n'est généralement pas déterministe. Elle dépend de la rapidité des liens de communication, mais également de l'état initial du système. La complexité au pire est la mesure de la phase de stabilisation la plus longue, et cela en tenant compte de toutes les exécutions possibles pour chacun des états initiaux.
   (b) **Complexité en moyenne :** cette mesure est peu utilisée car, outre le fait qu'elle soit particulièrement délicate à établir, les diverses définitions qu'on en trouve à travers la littérature ne concordent pas toujours entres elles. Pour certains, elle est la moyenne des tailles de toutes les phases de stabilisation, en prenant en compte tous les états initiaux et exécutions possibles. D'autre pondèrent le poids des exécutions en leur affectant des probalités.

L'importance de ces critères est néanmoins à relativiser. En effet, ils sont à mettre en regard avec le jeu d'hypothèses nécessaires au bon fonctionnement de l'algorithme évalué. Par exemple, si un algorithme autorise des processeurs distants à communiquer directement entre eux, l'évaluation de sa complexité en temps est biaisée car en cas d'implantation, cette communication à distance se traduirait par un surcoût généralement non négligeable.

# Chapitre 2

# Modèle

*Le modèle. Tout est dans le modèle."*
Yves Saint Laurent

Le but de ce chapitre est de formaliser tous les concepts introduits dans le chapitre précédent. La modèlisation fournit un cadre permettant de présenter des algorithmes répartis et de démontrer leurs propriétés. Tout au long de cette thèse, nous utiliserons principalement le modèle classique que la littérature appelle "à état". C'est un modèle à mémoire partagée utilisant des registres atomiques.

Dans une première partie, nous rappelons quelques définitions globales sur les graphes et autres objets classiques utilisés en systèmes répartis. Puis nous définissons de manière précise l'auto-stabilisation, la $k$-stabilisation, la stabilisation proportionnelle et la stabilisation anti-corruption. Enfin, nous présentons les outils classiquement utilisés pour prouver les propriétés des algorithmes répartis.

## 2.1 Exécutions

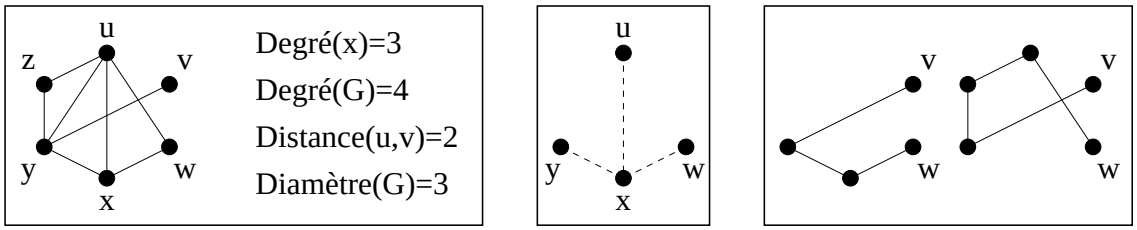
### 2.1.1 Graphe

Dans un système réel, les processeurs communiquent entre eux grâce à des liens de communication. La topologie du réseau auquel ils appartiennent est modélisé par un graphe.

**Définition 2.1.1 : Graphe**
Un *graphe* est un couple $(S,A)$ avec $S$ ensemble des nœuds (aussi appelé sommets) du graphe et $A \in S \times S$ ensemble des liens entre les differents nœuds.

Quelques exemples de graphe ont déjà été présentés figure 1.2, page 18. Un autre exemple est donné figure 2.1.(a).

Selon que le réseau est bidirectionnel ou non, on peut utiliser un graphe orienté ou non-orienté. De même, pour modéliser un réseau semi-uniforme, on peut considérer un graphe avec racine :




(a) Graphe G (b) Voisinage de x (c) 2 chemins allant de v à w

FIG. 2.1 – *Outillage basique des graphes*

**Définition 2.1.2 : Divers graphes**
Un graphe est *orienté* si les couples composant $V$ sont ordonnés ($(u,v)$ et $(v,u)$ sont deux liens distincts). Dans ce cas, les liens entre les nœuds sont appelés *arcs* du graphe.
Un graphe est *non orienté* si les couples composant $V$ ne sont pas ordonnés ($(u,v) = (v,u)$). Dans ce cas, les liens sont appelés *arètes*.
Un graphe *enraciné* (orienté ou non) est un triplet $(S,V,R)$ avec $S$ ensemble des nœuds, $V \in S \times S$ ensemble des liens reliant les différents nœuds et $R$ un nœud de $S$. $R$ est appelé *racine* du graphe.

Il existe un grand nombre d'autres graphes, par exemple des graphes avec poids où à chaque lien est associé un réel positif, ou encore des graphes étiquetés. Par la suite et sauf mention contraire, nous ne considèrerons que des graphes non orienté.

Dans ce qui suit, on considère un graphe $G = (S,A)$ et deux de ses nœuds $u$ et $v$.

**Définition 2.1.3 : Voisinage**
$u$ est un *voisin* de $v$ s'il existe un lien reliant $u$ à $v$. Le *voisinage de u* (noté $\mathcal{N}(u)$) est l'ensemble de tous les voisins de $u$ : $\mathcal{N}(u) = \{v : (u,v) \in A\}$.

Toujours en prenant l'exemple du graphe $G$ figure 2.1.(a), le voisinage du nœud $x$ est l'ensemble des points $\{v,y,w\}$.

**Définition 2.1.4 : Degré**
Le *degré d'un nœud* est le nombre de ses voisins :
$$Degre(u) = Cardinal\{Voisinage(u)\}$$
Le *degré d'un graphe* est le plus grand degré de ses nœuds :
$$Degre(G) = Max_{u \in S}\{Degre(u)\}$$

Sur l'exemple figure 2.1.(a), le degré du nœud $x$ est 3 alors que le dégré du graphe est 4 (car $y$ a un degré de 4).

**Définition 2.1.5 : Chemin / connexité**
Un *chemin* entre $u$ et $v$ est une suite de nœuds, chacun voisin de son prédécesseur, allant de $u$ à $v$.
$(u_0,u_1,...,u_i)$ est un chemin de $u$ a $v$ si $u = u_0$, $v = u_i$ et $\forall j \in [1..i]$, $(u_{j-1},u_j) \in A$.

Un graphe est *connexe*[1]si deux nœuds quelconques du graphe sont toujours reliés par un chemin.

Deux exemples de chemins reliant $v$ à $w$ sont donnés figure 2.1.(c).

**Définition 2.1.6 : Distance**
La *distance* entre deux nœuds $u$ et $v$ distincts[2]est le nombre d'éléments composant le plus court chemin allant de $u$ à $v$ (en incluant $u$ et en excluant $v$).
$$Distance(u,v) = Min_{C \in Chemin(u,v)}\{Cardinal\{C\} - 1\}$$

**Définition 2.1.7 : Dimension d'un graphe**
Le *diamètre* du graphe $G$ est la plus grande distance séparant deux nœuds de $G$.
$$Diametre(G) = Max_{\{u,v\} \in S \times S} Distance(u,v)$$

La *profondeur* du graphe enraciné $G = (S,A,R)$ est la plus grande distance séparant un nœud de la racine $R$.
$$Prof(G) = Max_{u \in S} Distance(u,R)$$

Toujours figure 2.1.(a), la distance entre $u$ et $v$ est de 2 (le chemin $(u,y,v)$ relie $u$ à $v$ et est de longueur 2). Le diamètre de $G$ est de 3 (la distance entre $v$ et $w$ est de 3).

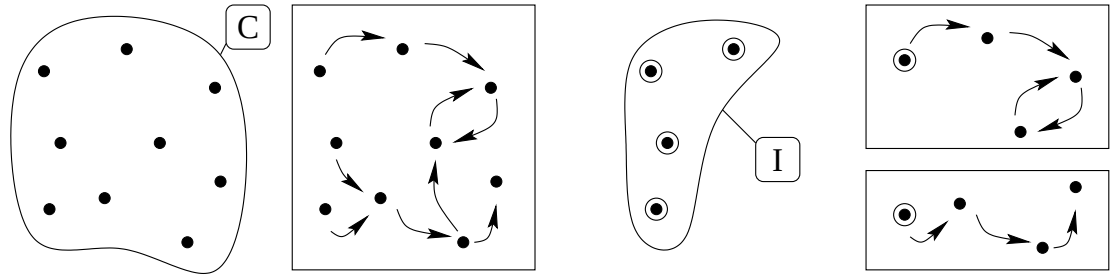
### 2.1.2 Systèmes répartis dans le modèle à état

Classiquement, pour modéliser les exécutions des systèmes répartis, on utilise la notion de système de transitions.

Un système de transition est composé de l'ensemble de toutes les configurations possibles du système réparti, d'une relation de transition permettant de définir l'évolution du système et d'un ensemble d'états pouvant servir d'états initiaux aux exécutions.

**Définition 2.1.8 : Système de transitions**
Un *système de transition* $\mathcal{S}$ est un triplet $(\mathcal{C}, \longrightarrow, \mathcal{I})$. $\mathcal{C}$ est un ensemble, $\longrightarrow$ est une relation binaire (étiqueté) sur $\mathcal{C}$, et $\mathcal{I}$ est un sous-ensemble de $\mathcal{C}$.
Un *système de transition étiqueté* $\mathcal{S}$ est un triplet $(\mathcal{C}, \stackrel{*}{\longrightarrow}, \mathcal{I})$. $\mathcal{C}$ est un ensemble, $\stackrel{*}{\longrightarrow}$ est une relation binaire étiqueté sur $\mathcal{C}$, et $\mathcal{I}$ est un sous-ensemble de $\mathcal{C}$.

1. Formellement, cette définition est celle de la *connexite par arc*. Mais dans le cas d'un graphe fini, les notions de *connexité* et *connexité par arc* sont identiques.
2. La distance d'un nœud à lui-même est zero.




(a) Configurations C (b) Transitions (c) Configurations initiales I (d) Deux exécutions

FIG. 2.2 – *Exemple de système de transition*

**Définition 2.1.9 : Transition**
Etant donné un système de transition $\mathcal{S}$, une *transition* (respectivement *transition étiqueté*) $T$ est un couple $(C_1,C_2) \in \mathcal{C} \times \mathcal{C}$ appartenant à la relation de transition $\longrightarrow$ (resp. $\stackrel{*}{\longrightarrow}$) .

On note $T = C_1 \longrightarrow C_2$ (resp . $T = C_1 \stackrel{*}{\longrightarrow} C_2$

*L'origine* de la transition $T$ est la configuration $C_1$. *L'arrivée* de $T$ est la configuration $C_2$.

$$Origine(C_1 \longrightarrow C_2) = C_1$$
$$Arrivee(C_1 \longrightarrow C_2) = C_2$$

**Définition 2.1.10 : Configuration terminale**
Soit $\mathcal{S}$ un système de transition et $C_1$ un élément de $\mathcal{C}$. $C_1$ est une *configuration terminale* de $\mathcal{C}$ si aucune transition n'a $C_1$ pour origine :

$$C_1 \text{ terminale} \Longleftrightarrow \nexists C_2 \text{ tel que } C_1 \longrightarrow C_2$$

Un exemple de système de transition est présenté figure 2.2. Le système de transition $\mathcal{S}$ est constitué d'un ensemble de configurations $C$, de transitions et d'un ensemble de configurations initiales $I$.

**Définition 2.1.11 : Exécution**
Une *exécution* $\mathcal{E}$ du système de transition $\mathcal{S}$ est une suite maximale de transition $\mathcal{E} = (C_0 \longrightarrow C_1, C_1 \longrightarrow C_2, C_2 \longrightarrow C_3, ...)$ tel que $C_0$ soit dans l'ensemble des configurations initiales $\mathcal{I}$. Par simplicité, les exécutions sont également notées $\mathcal{E} = (C_0 \longrightarrow C_1 \longrightarrow C_2 \longrightarrow C_3 ...)$ ou encore $\mathcal{E} = (C_0, C_1, C_2, C_3, ...)$. Une exécution peut être finie ou infinie. Une *exécution partielle* est un préfixe non vide d'une exécution.

Deux exécutions possibles du système de transition $\mathcal{S}$ sont représentées figure 2.2.(d). La première est infinie (deux transitions puis une boucle) ; le seconde est finie (trois transitions).

**Algorithme réparti**

Dans un système réparti, la plus petite unité de travail considéré est le processeur.

Informellement, un processeur est une unité de calcul disposant d'une mémoire et capable d'effectuer un certain nombre de tâches. Dans tous les cas, il peut effectuer des opérations internes, comme lire le contenu de ses variables ou évaluer des expressions arithmétiques. Via le réseau, il peut également communiquer avec d'autres processeurs.

Formellement, un processeur est modélisé par une machine à état.

**Définition 2.1.12**
Une machine à états, ou automate, est un quadruplet $(Z,E,T,I)$ où :

1. $Z$ est un ensemble d'états.
2. $E$ est un ensemble d'actions.
3. $T$ est une fonction qui à un couple (état,action) associe un autre état : $T : Z \times E \longmapsto Z$.
4. $I$, sous ensemble de $Z$, est l'ensemble des états initiaux.

Un processeur exécutant un code est alors modélisé par l'automate $(Z_P,E_P,T_P,I_P)$ suivant :

1. $Z_P$, ensemble des états de $P$, est l'ensemble de toutes les valeurs possibles des variables de $P$. Par commodité, nous considèrerons l'ensemble des variables de $P$ comme une seule entité que nous appellerons *la* variable de $P$.
2. $E_P$ est l'ensemble de toutes les actions possibles de $P$. Cela inclut d'éventuelles actions de lecture des variables d'un autre processeur. Dans ce cas, la machine à état prend en compte tous les résultats possibles de la lecture.
3. Si $e \in E_P$ est une action et $z \in Z_P$ un état, $T_P(z,e)$ est l'état que $P$ atteint à partir de l'état $z$ par l'exécution de l'action $e$.
4. $I$ est l'ensemble des états initiaux autorisé pour $P$.

Trois actions jouent pour les processeurs un rôle particulier : l'action vide (noté $*$), signifie que l'automate ne fait rien. Si $E$ est un ensemble d'actions, on note $E^*$ l'ensemble constitué des actions de $E$ union l'action vide : $E^* = E \cup \{*\}$.

L'action d'écriture correspond à une opération d'affectation. Après l'action $Ecriture_P(x)$, la valeur de la variable de $P$ vaut $x$.

L'action de lecture modélise la réception d'un message. Plus précisement, l'action $Lecture_P(Q,x)$ simule la reception du message "la valeur de la variable du processeur $Q$ est $x$".

**Définition 2.1.13**
Un algorithme distribué $\mathcal{A}$ pour un groupe de processeurs $\{P_i\}_{1\leq i\leq n}$ est un ensemble d'automates pour chacun des $P_i$ :

$$\mathcal{A} = \{A_1,A_2,...,A_n\}$$

**Lien entre "système de transition" et "algorithme distribué"** Les exécutions des algorithmes distribués peuvent être modélisés grâce à un système de transition. Dans

le cas particulier d'un algorithme où il n'y aurait pas d'interaction entre les divers prcesseurs, le système de transition est obtenu en faisant le produit cartésien des automates de tous les processeurs :

**Définition 2.1.14**
Le produit cartésien d'un ensemble d'automate $\{(Z_i,E_i,T_i,I_i)\}_{1\leq i\leq n}$ est un automate $(Z,E,T,I)$ tel que :

1. $Z = Z_1 \times Z_2 \times ... \times Z_n$
2. $E = E_1^* \times E_2^* \times ... \times E_n^*$
3. Si $z = (z_1,...,z_n) \in Z$ et $e = (e_1,...,e_n) \in E$, $T(z,e)$ est la configuration $z' = (z'_1,...z'_n)$ avec pour tout i compris entre 1 et $n$, $z'_i = T_i(z_i,e_i)$
4. $I = I_1 \times I_2 \times ... \times I_n$

A partir de là, on peut alors construire le système de transition étiqueté $\mathcal{S} = (\mathcal{C}, \longrightarrow, \mathcal{I})$ associé à l'algorithme distribué $\mathcal{A} = \{A_1,A_2,...,A_n\}$ contruisant le produit cartésien des automates $(Z,E,T,I)$, puis en posant :

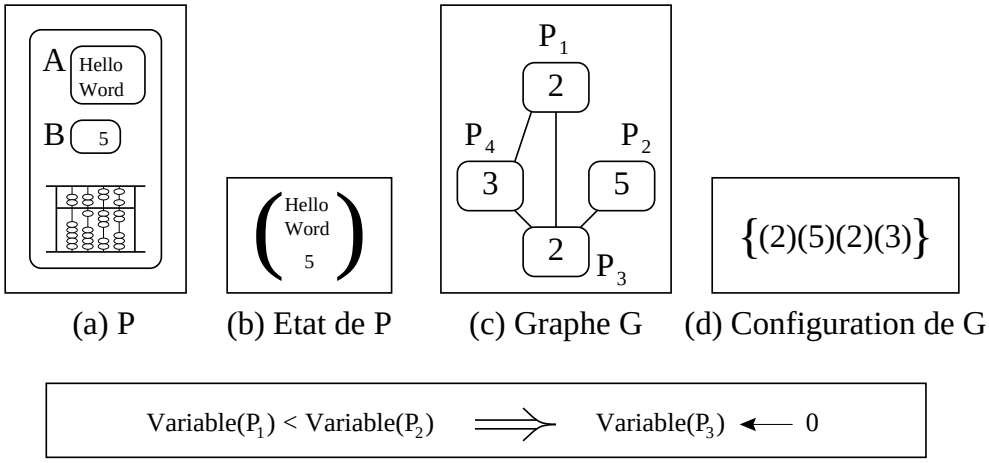
1. $\mathcal{C} = Z$
2. $C \xrightarrow{e} C'$ est une transition étiquetée s'il existe $e$ dans $E-\{*\}$ tel que $C' = T(C,e)$.
3. $\mathcal{I} = I$

**Notation :** $\mathcal{C}$ est l'ensemble des configurations du système. Etant donné une configuration $C$ de $\mathcal{C}$, on note $Z_C(P)$ l'état de $P$ dans la configuration $C$.

Dans le cas plus général où les divers processeurs peuvent interagir entre eux, les exécutions des algorithmes distribués sont également modélisées par un système de transitions. Il utilise toujours pour support le produit cartésien des automates de tous les processeurs. Mais la relation de transition est restreinte à certaines actions composées. Dans le cas du modèle à états, deux contraintes sont à considérer.

1. Si un processeur $P_i$ effectue une opération de lecture de variable du processeur $P_j$, alors la valeur lue par $P_i$ doit correspondre à la valeur effective de la variable de $P_j$.
2. Une même variable ne peut pas faire l'objet simultanément d'une opération de lecture et d'un opération d'écriture.

On peut ensuite constuire un système de transition associé à un algorithme distribué de la manière suivante :




(a) P (b) Etat de P (c) Graphe G (d) Configuration de G

(e) Exemple de règle gardé pour $P_3$

FIG. 2.3 – *Outillage basique des systèmes répartis*

**Validation 2.1.15**
Soit $\mathcal{A} = \{A_1,A_2,...,A_n\}$ un algorithme distribué et $(Z,E,T,I)$ le produit cartésien des automates de $\mathcal{A}$. On construit le système de transition étiqueté $\mathcal{S} = (\mathcal{C}, \longrightarrow, \mathcal{I})$ associé a $\mathcal{A}$ en posant :

1. $\mathcal{C} = Z$
2. $C \longrightarrow C'$ est une transition s'il existe $e$ dans $E$ privé de l'action nulle vérifiant :
   (a) $C' = T(C,e)$.
   (b) Il n'existe pas $j$ et $j'$ tel que $e_j = Lecture_P(Q,x)$ et $e_{j'} = Ecriture_Q(y)$.
   (c) Si $e_j = Lecture_P(Q,x)$, alors $x = Z_C(Q)$.

   $e$ est alors l'étiquette de $\longrightarrow$.
3. $\mathcal{I} = I$

### 2.1.3 Pratiquement...

Le formaliste que nous venons de décrire nous donne un cadre permettant de valider les propriétés d'un algorithme. Néanmoins, la donné d'un algorithme réparti sous forme d'automate peut s'avérer lourde. Aussi utilise-t-on une écriture simplifié basé sur la notion de règle gardés.

**Définition 2.1.16Etat**
*L'état* d'un processeur $P$, noté $Etat(P)$ est l'ensemble des valeurs prise par les variables du processeur.
La configuration d'un ensemble de processeurs est l'ensemble des états de tous les processeurs.

Une illustration est donnée figure 2.3. L'état de $P$ (figure 2.3(b)) est le contenu de ses variables. Le graphe $G$ figure 2.3.(c), est consitué de 4 processeurs $P_1$, $P_2$, $P_3$ et $P_4$. Sur notre exemple, chaque processeur dispose d'une variable. Une configuration de $G$ est l'ensemble des valeurs de tous les processeurs de $G$ (figure 2.3.(d)).

**Notation :** étant donné une configuration $C$, l'état de $P$ dans la configuration

**Définition 2.1.17 : Règle gardée**
Soit $P$ un processeur. Une *règle gardée* pour $P$ est un couple ($C$,$T$) où $C$ est une condition et $T$ une action d'action.
Pour un processeur donné, un ensemble de règles gardées est disjoint si deux conditions ne peuvent être simultanément satisfaites.

Dans le modèle à état, la condition (également appelée *garde*) est une expression booléenne portant sur l'état de $P$ et de ses voisins. L'action de $P$ est une action d'écriture dans une ou plusieurs variables de $P$.

Pour évaluer la condition d'une règle gardée, un processeur doit lire le contenu de toutes les variables apparaissant dans la condition. On dit qu'il évalue sa garde. Dans le modèle à état, on considére qu'une action de lecture de $P$ lui permet de prendre connaissance des états de tous ses voisins.

Un algorithme distribué pour un ensemble de processeurs $\{P_i\}$ est alors la donnée d'un ensemble de régles gardées pour chacun des $P_i$.

**Définition 2.1.18 : Algorithme réparti**
Soit $\mathcal{P}$ un ensemble de processeurs. Un *algorithme réparti* $\mathcal{A}$ pour l'ensemble de processeurs $\mathcal{P}$ est une fonction associant à chaque processeur de $\mathcal{P}$ un ensemble de règles gardées.

### 2.1.4 Pseudo-code

Dans les chapitres suivants, nous allons présenter un certain nombre d'algorithmes répartis. Nous venons de le voir, un algorithme réparti est une collection de règles gardées associées à chacun des processeurs. Une règle gardée est un couple (*condition*,*action*). Nous présentons les algorithmes de cette thèse en donnant la liste des prédicats utilisés par les processeurs, puis les actions qu'ils peuvent accomplir et enfin les règles gardées.

**Prédicats :** un prédicat est une condition booléenne. Si $Cond(P,\mathcal{N}(P))$ est une condition booléenne portant sur les variables de $P$ et celle de ses voisins, on note $Predicat(P) \Leftrightarrow Cond(P,\mathcal{N}(P))$.

**Action :** dans le modèle à états, les seules actions autorisées sont des opérations d'écriture, c'est à dire d'affectation d'une valeur à une variable. Si $P$ affecte $x$ à la variable $Val(P)$, on note $Val(P) \longleftarrow x$. Si cette action d'affectation porte le nom de $AffecteVal(P)$, on note $AffecteVal(P) \Leftrightarrow Val(P) \longleftarrow x$.

**Règles gardées :** les règles gardées sont une combinaison des différentes conditions et actions précédemment définies. Toujours sur le même exemple, si $(Cond(P,\mathcal{N}_P), Val(P) \longleftarrow x)$ est une règle gardée, elle est notée $Predicat(P) \Longrightarrow AffecteVal(P)$.

Un exemple d'algorithme illustrant ces notations est présenté figure 2.4

**Définition des prédicats**

$Pred1(P) \quad \Leftrightarrow Variable(P) = 0$

$Pred2(P) \quad \Leftrightarrow Variable(P) \neq 0$

**Définition des actions**

$Action1(P) \quad \Leftrightarrow Variable(P) \longleftarrow 0$

$Action2(P) \quad \Leftrightarrow Variable(P) \longleftarrow 1$

**Règles gardées**

$R1 \quad Pred1(P) \quad \Longrightarrow Action1(P)$

$R2 \quad Pred2(P) \quad \Longrightarrow Action2(P)$

FIG. 2.4 – *Pseudo code pour les algorithmes*

**Convention sur la présentation des algorithmes :** dans certain cas, il peut être lourd de noter exactement l'ensemble des conditions devant être vraies pour qu'une règle soit exécutée. Par exemple, si les deux conditions $Cond1$ et $Cond2$ peuvent être satifaites simultanément, un algorithme correctement écrit doit considérer tous les cas possibles : si $Cond1$ et $Cond2$ sont vrais, si seulement $Cond1$ est vrai, si seulement $Cond2$ est vrai et si $Cond1$ et $Cond2$ sont faux tous les deux.

En pratique, nous simplifierons cette écriture (lorsque c'est possible) en présentant les règles gardées sous forme non disjointes avec pour convention qu'un processeur activé par le démon doit exécuter simultanémant toutes ses règles activables (il doit lire toutes ses variables et celles de ses voisins, puis exécuter toutes les actions possibles). Si un processeur $P$ dispose de trois variables $V_1$, $V_2$ et $V_3$, l'exemple donné figure 2.5.(b) est une simplification de l'algorithme écrit formellement figure 2.5.(a).

### 2.1.5 Démon

Dans la section 2.1.2, nous avons définit l'ensemble des exécutions possibles d'un algorithme. Dans la pratique, cet ensemble est généralement trop vaste pour permettre de prouver les propriétés que l'on cherche à établir. D'où l'introduction d'une hypothèse simplificatrice appelée *le démon*.

Du point de vue théorique, le démon est une restriction de l'ensemble des exécutions possibles d'un système de transition.

**Définition 2.1.19 : Démon**
Un *démon* est un prédicat portant les exécutions des systèmes de transition.

Le démon sert à restreindre l'emsemble de toutes les exécutions possibles. Lorsque l'on considère un système de transitions, les exécutions ne vérifiant pas le prédicat du

**Règles gardées**

| | | |
|---|---|---|
| $R1$ | $V_1(P) = 0$, $V_2(P) = 0$ et $V_3(P) = 0$ | $\Longrightarrow Action1(P)$, $Action2(P)$ et $Action3(P)$ |
| $R2$ | $V_1(P) = 0$, $V_2(P) = 0$ et $V_3(P) \neq 0$ | $\Longrightarrow Action1(P)$ et $Action2(P)$ |
| $R3$ | $V_1(P) = 0$, $V_2(P) \neq 0$ et $V_3(P) = 0$ | $\Longrightarrow Action1(P)$ et $Action3(P)$ |
| $R4$ | $V_1(P) = 0$, $V_2(P) \neq 0$ et $V_3(P) \neq 0$ | $\Longrightarrow Action1(P)$ |
| $R5$ | $V_1(P) \neq 0$, $V_2(P) = 0$ et $V_3(P) = 0$ | $\Longrightarrow Action2(P)$ et $Action3(P)$ |
| $R6$ | $V_1(P) \neq 0$, $V_2(P) = 0$ et $V_3(P) \neq 0$ | $\Longrightarrow Action2(P)$ |
| $R7$ | $V_1(P) \neq 0$, $V_2(P) \neq 0$ et $V_3(P) = 0$ | $\Longrightarrow Action3(P)$ |

(a) Lourd

**Règles gardées**

| | | |
|---|---|---|
| $R1$ | $V_1(P) = 0$ | $\Longrightarrow Action1(P)$ |
| $R2$ | $V_2(P) = 0$ | $\Longrightarrow Action2(P)$ |
| $R2$ | $V_3(P) = 0$ | $\Longrightarrow Action3(P)$ |

(b) Light

FIG. 2.5 – *Simplification du seudo code*

démon ne sont plus à prendre en compte.

**Définition 2.1.20**
Soit $\mathcal{S}$ un système de transition et $\mathcal{D}$ un démon. L'ensemble des exécutions de $\mathcal{S}$ selon le démon $\mathcal{D}$ est l'ensemble des exécutions de $\mathcal{S}$ pour lesquelles $\mathcal{D}$ est vrai :

$$\text{Exécution de } \mathcal{S} \text{ selon } \mathcal{D} = \{\mathcal{E} : \mathcal{D}(\mathcal{E}) \text{ soit vrai.}\}$$

Dans la pratique, les démons sont souvent présentés comme un ensemble de condition que les exécutions doivent respecter pour être valides. En particulier, ils imposent généralement des conditions sur l'enchaînement des transitions. Par exemple, si $P1$ et $P2$ sont des processeurs (potentiellement confondus) et que $A1$ et $A2$ sont des actions, un démon peut imposer qu'une transition du type $C_1 \stackrel{P1,A1}{\longrightarrow} C_2$ soit toujours suivie d'une transition $C_2 \stackrel{P2,A2}{\longrightarrow} C_3$. Dans ce cas, les transitions qui doivent nécessairement se suivre sont regroupées sous une seule. Dans notre exemple, $C_1 \stackrel{P1,A1}{\longrightarrow} C_2 \stackrel{P2,A2}{\longrightarrow} C_3$ serait notée $C_1 \stackrel{(P1,A1)(P2,A2)}{\longrightarrow} C_3$ (ou $C_1 \stackrel{(P1,P2)}{\longrightarrow} C_3$, ou encore $C_1 \longrightarrow C_3$ s'il n'y a pas d'ambiguité). Formellement, la concaténation de plusieurs transitions indissociables est appelée *méta-transition*.

**Définition 2.1.21**
Une méta-transition définie selon un démon $\mathcal{D}$ est une suite de transitions $(C_0 \longrightarrow C_1 \longrightarrow C_2 \ \ldots \ C_{i-1} \longrightarrow C_i)$ tel que dans toute exécution validé par le démon, la transition $C_0 \longrightarrow C_1$ est toujours suivie par les $i-1$ transitions $(C_1 \longrightarrow C_2 \ \ldots \ C_{i-1} \longrightarrow C_i)$.

Vu sous cet angle, un démon peut alors être considéré comme un prédicat fixant l'atomicité des actions du système et éventuellement leur ordre, définissant ainsi la relation de méta-transition $\longrightarrow$. A travers la littérature, c'est principalement sous cette forme que les démons sont rencontrés. Leur définition fait état de la séquence d'actions élémentaires devant être contenues dans une méta-transition.

Certains démons devenus classiques portent des noms. Dans cette thèse, nous utilisons le démon *centralisé* et le démon *synchrone*.

Un démon centralisé fait agir les processeurs un par un et considère l'ensemble "lecture / écriture" comme atomique.

**Définition 2.1.22 : Démon centralisé**
Soit $\mathcal{S}$ un système de transition. Une méta-transition sous un démon centralisé est une suite de deux transitions, la première étant une opération de lecture d'un processeur $P$, la seconde une opération d'écriture du même processeur $P$, les deux opérations faisant partie de la même règle gardée.

Un démon synchrone fait agir tous les processeurs simultanément. Plus précisément, tous les processeurs ayant des règles activables doivent effecuer simultanément leurs opérations de lecture, puis leurs opérations d'écritures.

**Définition 2.1.23 : Démon synchrone**
Soit $\mathcal{S}$ un système de transition et $\mathcal{E}$ une exécution de $\mathcal{S}$. Une méta-transition $T = C_1 \longrightarrow C_2$ sous un démon synchrone est une suite de transitions vérifiant :

1. $T$ contient une opération de lecture de tous les processeurs activables dans $C_1$.
2. $T$ contient une opération d'écriture de tous les processeurs activables dans $C_1$.
3. Toutes les opérations d'écriture de $T$ ont lieu après les opérations de lecture.
4. Les opérations de lecture et d'écriture d'un processeur appartiennent à la même règle gardée.

A travers la littérature, les méta-transitions portent des noms différents selon le démon qui les définis : dans le cas d'un démon synchrone, les méta-transitions sont généralement appelées *rounds*. Dans le cas d'un démon centralisé, elles sont par abus de langage appelées *transitions*.

## 2.2 Algorithmes stabilisants

Dans les systèmes répartis classiques, l'ensemble des exécutions est dans une large mesure restreint par les fortes contraintes imposées aux configurations initiales. En effet, $\mathcal{I}$ peut ne contenir que des configurations à partir desquelles les exécutions vérifient les spécifications du problème. Les algorithmes stabilisants sont moins restrictifs et autorisent des ensembles de configurations initiales plus vastes.

Plus précisément, étant donné un système de transitions, tous les algorithmes stabilisants satisfont une même propriété : il existe un ensemble de configurations dites légitimes vérifiant la convergence et la correction. La convergence est l'assurance que toute exécution atteindra une configuration légitime. La correction est la garantie que toute execution ayant une configuration initiale légitime satisfait la spécification du problème.

Avoir les deux propriétés ensemble assure que toutes les exécutions vont atteindre une configuration légitime à partir de laquelle la spécification du problème sera vérifié.

Les différents raffinements de la stabilisation présentés ci-dessous portent sur l'ensemble des configurations initiales et sur le temps de stabilisation.

### 2.2.1 Configurations initiales

Les algorithmes auto-stabilisants ou proportionnels acceptent des configurations initiales quelconques. Par contre, les algorithmes $k$-stabilisants et $k$-proportionnels exigent des configurations initiales faiblements corrompues.

Informellement, on définit les configurations "partiellement corrompues" relativement à des configurations non corrompues, appelées configurations correctes :

**Définition 2.2.1 : Configuration correcte**
Soit $\mathcal{S}$ un système et $\mathcal{SP}$ la spécification d'un problème. Une configuration $L$ est *correcte* si toute exécution du système ayant $L$ pour configuration initiale vérifie $\mathcal{SP}$. L'ensemble de toutes les configurations correctes est noté $\mathcal{CC}$.

Une configuation partiellement corrompue est alors une configuration "presque" correcte. Formaliser le "presque" nécéssite l'introduction de la distance de Hamming entre configurations et entre ensembles de configurations :

**Définition 2.2.2 : Distance de Hamming (entre configurations)**
Etant donné un graphe $G = (S,V)$, la *distance* entre deux configurations $L_1$ et $L_2$ est le nombre de processeurs n'ayant pas les mêmes valeurs dans $L_1$ et $L_2$.

$$Dist(L_1,L_2) = Cardinal\{P \in S : Etat_{L_1}(P) \neq Etat_{L_2}(P)\}$$

La distance entre $L_1$ et un ensemble de configurations $\mathcal{L}_2$ est alors la plus petite distance entre $L_1$ et une des configurations de $\mathcal{L}_2$.

$$Dist(L_1,\mathcal{L}_2) = Min_{L_2 \in \mathcal{L}_2}\{Dist(L_1,L_2)\}$$

A partir de là, la notion de *corruption partielle* est formalisable : une configuration est $k$-corrompue si sa distance à l'ensemble des configurations correctes $\mathcal{CC}$ est inférieur ou égale à $k$. On dit alors qu'elle est dans la boule de rayon $k$ et de centre $\mathcal{CC}$.

**Définition 2.2.3 : Boule**
La *boule* de centre $\mathcal{CC}$ et de rayon $k$ est l'ensemble des configurations dont la distance à $\mathcal{CC}$ est inférieure ou égale à $k$.

$$\mathcal{B}_k(\mathcal{CC}) = \{L : Dist(L,\mathcal{CC}) \leq k\}$$

Tous les algorithmes stabilisants vérifient la propriété de convergence. Certains la vérifient à partir d'un ensemble de configurations initiales quelconques, d'autres uniquement à partir d'un ensemble de configurations restreint à la boule de rayon $k$ ayant pour centre les configurations correctes. Dans ce second cas, la propriété de convergence sera appelée la $k$-convergence.

### 2.2.2 Temps de convergence

L'autre différence majeure entre les diverses formes d'algorithmes stabilisants porte sur le temps de convergence. Informellement, le temps de convergence (ou complexité en temps) est le nombre de transitions nécessaires au système avant qu'il atteigne une configuration correcte.

**Définition 2.2.4 : Temps de convergence**
Soit $\mathcal{S} = \{\mathcal{C}, \longrightarrow, \mathcal{I}\}$ un système de transition et $\mathcal{E}$ une exécution de $\mathcal{S}$. Soit $\mathcal{CC}$ l'ensemble des configurations correctes et $\mathcal{E}$ une exécution du système contenant au moins une configuration de $\mathcal{CC}$.
On appelle *temps de convergence de* $\mathcal{E}$ le nombre de configurations comprises entre la configuration initiale de $\mathcal{E}$ et la première configuration appartenant à $\mathcal{CC}$. On le note :

$$TempsConv(\mathcal{E}))$$

Le *temps de convergence (au pire) du système* $\mathcal{S}$ *vers* $\mathcal{L}$ est le plus grand temps de convergence de toutes les exécutions possibles.

$$TempsConv(\mathcal{S}) = Max_{\mathcal{E}}\{TempsConv(\mathcal{E})\}$$

En général, la compléxité est fonction d'un certain nombre de paramètres comme la taille du graphe ou le nombre initial de fautes.

### 2.2.3 Définition

#### A) Auto-stabilisation

[Del95] récapitule et compare les différentes définitions de l'auto-stabilisation que l'on peut trouver à travers la littérature. Nous adopterons pour notre part celle de Tell [Tel94] Comme nous l'avons déjà vue au chapitre 1, un système auto-stabilisant n'impose aucune contrainte sur le temps de convergence de l'algorithme, pas plus que sur l'ensemble des configurations initiales. Toutes les configurations peuvent être le départ d'une exécution.

**Définition 2.2.5 : Auto-stabilisation**
Un système de transition $\mathcal{S}$ est auto-stabilisant pour la spécification $\mathcal{SP}$ si les configurations initiales sont quelconques ($\mathcal{I} = \mathcal{C}$) et s'il existe $\mathcal{L}$, un sous-ensemble de $\mathcal{C}$ vérifiant :

1. **Correction :** toute exécution dont la configuration initiale est dans $\mathcal{L}$ vérifie la spécification $\mathcal{SP}$ (ou encore : $\mathcal{L}$ est un sous-ensemble de l'ensemble des configurations correctes).
2. **Convergence :** toute exécution contient une configuration de $\mathcal{L}$.

$\mathcal{L}$ est appelé ensemble des *configurations légitimes*.

#### B) $K$-stabilisation

Les systèmes $k$-stabilisants ont un temps de convergence quelconque mais imposent certaines restrictions sur leur ensemble de configurations initiales.

**Définition 2.2.6 : $K$-stabilisation**
Soit $k$ en entier. Un système de transition $\mathcal{S}$ est $k$-stabilisant pour la spécification $\mathcal{SP}$ s'il existe $\mathcal{L}$ (appelé ensemble des configurations légitimes), un sous-ensemble de $\mathcal{C}$, vérifiant :

1. **Correction :** toute exécution dont la configuration initiale est dans $\mathcal{L}$ vérifie la spécification $\mathcal{SP}$.
2. **$K$-convergence :** toute exécution ayant une configuration initiale dans $\mathcal{B}_k(\mathcal{CC})$ contient une configuration de $\mathcal{L}$.

### C) Stabilisation proportionnelle

Les algorithmes proportionnel sont des algorithmes auto-stabilisants ayant un temps de stabilisation proportionnel au nombre effectif de fautes frappant le réseau.

**Définition 2.2.7 : Stabilisation proportionnelle**
Un système de transition $\mathcal{S}$ est proportionnel pour la spécification $\mathcal{SP}$ s'il existe un polynome $P$ de $\mathbb{N}$ dans $\mathbb{R}$ et $\mathcal{L}$ (appelé ensemble des configurations légitimes), un sous-ensemble de $\mathcal{C}$, vérifiant :

1. **Correction :** toute exécution dont la configuration initiale est dans $\mathcal{L}$ vérifie la spécification $\mathcal{SP}$.
2. **Convergence :** toute exécution contient une configuration de $\mathcal{L}$.
3. **Complexité en temps proportionnelle :** pour toute exécution $\mathcal{E}$, si $f$ est la distance entre la configuration initiale de $\mathcal{E}$ et l'ensemble des configurations correcte, $\mathcal{E}$ a un temps de convergence de $P(f)$.

### D) Stabilisation $K$-proportionnelle

De la même manière, les algorithmes $k$-proportionnels sont des algorithmes $k$-stabilisants ayant un temps de stabilisation proportionnel au nombre effectif de fautes frappant le réseau.

**Définition 2.2.8 : $K$-proportionnelle**
Un système de transition $\mathcal{S}$ est $k$-proportionnel pour la spécification $\mathcal{SP}$ s'il existe un polynome $P$ de $\mathbb{N}$ dans $\mathbb{R}$ et $\mathcal{L}$ (appelé ensemble des configurations légitimes), un sous-ensemble de $\mathcal{C}$, vérifiant :

1. **Correction :** toute exécution dont la configuration initiale est dans $\mathcal{L}$ vérifie la spécification $\mathcal{SP}$.
2. **$K$-convergence :** toute exécution ayant une configuration initiale dans $\mathcal{B}_k(\mathcal{CC})$ contient une configuration de $\mathcal{L}$.
3. **Complexité en temps proportionnelle :** pour toute exécution $\mathcal{E}$, si $f$ est la distance entre la configuration initiale de $\mathcal{E}$ et l'ensemble des configurations correctes, $\mathcal{E}$ a un temps de convergence de $P(f)$.

**E) Stabilisation anti-corruption**

Dans la majorité des algorithmes, on distingue deux types de variables : les variables de sorties sont celles qui sont lues pour déterminer si l'algorithme répond à sa spécification. Ce sont elles qui donnent ou non les résultats escomptés. Généralement, elles ne sont pas les seules variables mises en jeu au cours d'une exécution. Les contrôles ou échanges d'information effectués par les processeurs pour parvenir à résoudre le problème nécessitent bien souvent l'utilisation de variables auxiliaires. Ces variables ont une importance moindre dans la mesure où si elles sont corrompues sans que leurs fausses valeurs modifient les variables de sorties, l'algorithme continue à avoir un comportement vérifiant la spécification du problème. Aussi la propriété anti-corruption ne porte que sur les variables de sortie.

**Définition 2.2.9 : Transition corrompant[3] un processeur**
Soit $\mathcal{S}$ un système de transition, $\mathcal{SP}$ la spécification d'un problème et $\mathcal{CC}$ l'ensemble des configurations correctes. Soit $T = C_1 \xrightarrow{\{P_i\}} C_2$ une transition et $P$ un des $\{P_i\}$. Soit $C_1'$ la configuration obtenue à partir de $C_1$ en donnant aux variables de sortie de $P$ les valeurs qu'elles ont dans $C_2$. $T$ est une transition corrompant $P$ si la distance entre $C_1$ et $\mathcal{CC}$ est plus petite que la distance entre $C_1'$ et $\mathcal{CC}$.

$$T \text{ corromp } P \iff Dist(C_1,\mathcal{CC}) < Dist(C_1',\mathcal{CC})$$

**Définition 2.2.10 : Anti-corruption**
Un système de transition $\mathcal{S}$ est anti-corruption pour la spécification $\mathcal{SP}$ s'il existe $\mathcal{L}$ (appelé ensemble des configurations légitimes), un sous-ensemble de $\mathcal{C}$, vérifiant :

1. **Convergence :** toute exécution contient une configuration de $\mathcal{L}$.
2. **Correction :** toute exécution dont la configuration initiale est dans $\mathcal{L}$ vérifie la spécification $\mathcal{SP}$.
3. **Anti-corruption :** il n'existe pas de transition corrompant un processeur.

## 2.3 Outils de démonstration

Démontrer qu'un algorithme est stabilisant se fait en plusieurs étapes. Tout d'abord, il convient de déterminer l'ensemble des configurations qui seront considérées comme légitimes. On montre ensuite les propriétés de correction puis de convergence. L'étude de la complexité (optionnelle pour la $k$-stabilisation et l'auto-stabilisation, mais néanmoins toujours faite) permet d'établir le caractère proportionnel d'un algorithme. Enfin, le cas échéant, l'anti-corruption conclut la preuve.

3. Dans [][A vérifier] définissent les transitions corruptrices comme étant des transitions corrompant tout le système. Nous présentons ici une version "faible" de la transition corruptrice puisqu'elle ne concerne qu'un seul processeur. Néanmoins, c'est bien du même concept qu'il s'agit.

### 2.3.1 Configurations légitimes

Tout d'abord, on doit définir un ensemble de configurations légitimes. Pour certains problèmes, notamment pour les problèmes statiques, un ensemble de configurations légitimes peut canoniquemment s'imposer : c'est l'ensemble des configurations correctes. Mais dans certain cas, il peut être intéressant (notamment pour simplifier des preuves) de restreindre l'ensemble des configurations légitimes à des configurations correctes ayant des propriétés spécifiques.

### 2.3.2 Correction

La correction est la propriété du système assurant qu'il répond bien à la spécification du problème.

Pour les problèmes dynamiques, la preuve de la correction est grandement similaire aux preuves classiques d'algorithmiques réparties : on suppose qu'une exécution a une configuration initiale légitime $L$. Puis on montre que toutes les exécutions partant de $L$ vérifient les spécifications du problème. Montrer cela pour $L$ quelconque prouve la correction.

On peut aussi montrer que l'ensemble des configurations légitimes est un sous-ensemble des configurations correctes. De par leur définition, les exécutions ayant des configurations correctes pour configuration initiale vérifient la correction. Il faut donc d'abord prouver qu'un ensemble est l'ensemble des configurations correctes, puis montrer que les configurations légitimes y sont inclues.

**Notation** Par simplicité, les exécutions dont la configuration initiale est légitime sont appelées exécutions légitimes.

**Définition 2.3.1**
Soit $\mathcal{S}$ un système de transition et $\mathcal{L}$ un ensemble de configuration légitime. Une *exécution légitime* est une exécution du système dont la configuration initiale est dans $\mathcal{L}$.

### 2.3.3 Convergence

La preuve de la convergence est souvent la partie la plus délicate de la démonstration. On utilise généralement trois types de techniques :

#### A) Fonction décroissante

Une manière de prouver la convergence consiste à montrer qu'il n'existe pas de configuration illégitime terminale (configuration à partir de laquelle aucune transition n'est possible, définition 2.1.10, page 34) et à exhiber une fonction $F$ ayant la propriété de décroître sur l'ensemble des configurations illégitimes.

Ce genre de fontion est appelée *fonction de convergence.*

**Convention :** selon les auteurs, le terme inférieur désigne la notion *d'inférieur strictement* ou celle *d'inférieur ou égale.* Dans tout ce qui suit, nous considérerons la première

définition. De même, les termes *supérieur*, *croissance* et *décroissance* feront toujours référence à des inégalités strictes.

**Définition 2.3.2 : Fonction décroissante**
Soit $\mathcal{S}$ un système réparti, $\mathcal{C}$ l'ensemble de toutes les configurations et $\mathcal{C}'$ un sous-ensemble de $\mathcal{C}$.
Soit $F$ une fonction de $\mathcal{C}$ dans un ensemble $\mathbb{N}$. $F$ est décroissante sur une transition $C_1 \longrightarrow C_2$ si la valeur de $F$ est plus petite sur $C_2$ que sur $C_1$.

$$C_1 \longrightarrow C_2 \implies F(C_1) > F(C_2)$$

$F : \mathcal{C}' \longrightarrow \mathbb{N}$ est décroissante sur $\mathcal{C}'$ si $F$ est décroissante sur toute transition impliquant deux configurations de $\mathcal{C}'$

$$\forall (C_1 \longrightarrow C_2) \in \mathcal{C}' \times \mathcal{C}', F(C_1) > F(C_2).$$

**Définition 2.3.3 : Fonction de convergence**
Soit $\mathcal{S}$ un système, $\mathcal{C}$ l'ensemble de toutes les configurations possibles et $\mathcal{L}$ un sous ensemble de $\mathcal{C}$. Une fonction $F$ de $\mathcal{C}$ dans $\mathbb{N}$ est une *fonction de convergence* vers $\mathcal{L}$ si elle est décroissante sur toute transition ne contenant pas de configurations de $\mathcal{L}$.

**Lemme 2.3.4[Tel94]**
Soit $\mathcal{S}$ un système réparti et $\mathcal{SP}$ une spécification. Soit $\mathcal{L}$ un ensemble de configurations légitimes. La convergence de $\mathcal{S}$ est assurée si :

1. il existe une fonction de convergence vers $\mathcal{L}$.
2. il n'existe pas de configuration terminale non légitime.

La preuve de ce lemme peut également être trouvée dans [Tel94].

**Preuve :** Soit $\mathcal{E}$ est une exécution du système et soit $\mathcal{E}' = (C_0, C_1, C_2, ...)$ l'exécution partielle de $\mathcal{E}$ ne comportant que des configurations non légitimes (éventuellement, $\mathcal{E}' = \mathcal{E}$). Considérons la suite $(F(C_0), F(C_1), F(C_2), ...)$. Elle est nécessairement strictement décroissante et positive, car $F$ est une fonction de convergence. Elle est donc finie. D'ou $\mathcal{E}'$ est aussi fini.

Soit $C_j$ la dernière configuration de $\mathcal{E}'$. Dans $\mathcal{E}$, $C_j$ n'est pas terminale car une configuration terminale non légitime n'existe pas. D'où $\mathcal{E}$ contient une transition $C_j \longrightarrow C_{j+1}$. Or, $C_{j+1}$ est légitime (sinon, elle serait dans $\mathcal{E}'$) et $\mathcal{E}$ converge. □

### B) Attracteur

Une fonction décroissante est parfois difficile à exhiber. Une variante consiste à définir des *attracteurs* d'exécution.

**Définition 2.3.5 : Attracteur**
Soit $\mathcal{S}$ un système réparti et $\mathcal{C}$ un ensemble de configurations. Soit $\mathcal{C}_1$ et $\mathcal{C}_2$ deux sous-ensembles de $\mathcal{C}$. $\mathcal{C}_2$ est un *attracteur* pour $\mathcal{C}_1$ si toute exécution du système contenant une configuration de $\mathcal{C}_1$ contient aussi une configuration de $\mathcal{C}_2$.

Intuitivement, définir une suite de $j$ attracteurs $\{\mathcal{A}_i\}$ avec $(1 \leq i \leq j)$ permet de

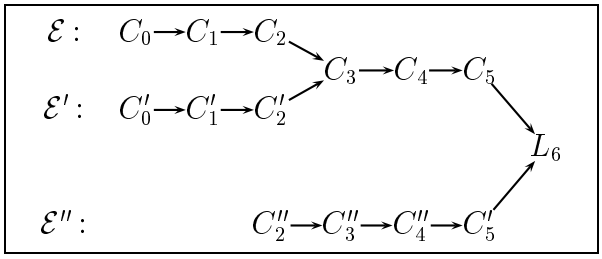



FIG. 2.6 – *Exemple d'exécutions C-équivalence*

morceler la preuve de la convergence : si $C_0$ est une configuration initiale, il peut être difficile de montrer qu'une exécution partant de $C_0$ contient toujours une configuration légitime. Il peut être plus aisé de prouver qu'elle contient une configuration appartenant à l'attracteur $\mathcal{A}_1$, puis qu'une exécution dont la configuration initiale est dans $\mathcal{A}_1$ contient une configuration dans l'attracteur $\mathcal{A}_2$ et ainsi de suite. Au final, si $\mathcal{A}_j$ est l'ensemble des configurations légitimes, on aura montré que toute exécution contient une configuration de $\mathcal{A}_1$, donc une configuration de $\mathcal{A}_2$, donc une configuration de $\mathcal{A}_3$ et ainsi de suite jusqu'à $\mathcal{A}_j$, l'ensemble des configurations légitimes.

**Théorème 2.3.6**
Soit $\mathcal{S}$ un système réparti et $\mathcal{SP}$ une spécification. Soit $\mathcal{L}$ un ensemble de configurations légitimes. La convergence de $\mathcal{S}$ est assurée s'il existe une suite $\{\mathcal{A}_i\}$ avec $(0 \leq i \leq j)$ de sous-ensembles de $\mathcal{C}$ vérifiant :

1. $\mathcal{A}_O = \mathcal{I}$, l'ensemble des configurations initiales.
2. $\mathcal{A}_j = \mathcal{L}$, l'ensemble des configurations légitimes.
3. Pour tout $i$ dans $[1..j]$, une exécution $\mathcal{E}$ dont la configuration initiale est dans $\mathcal{A}_{i-1}$ contient une configuration de $\mathcal{A}_i$.

**Preuve :** Soit $\mathcal{E}$ une exécution. Par définition, sa configuration initiale $C_0$ est dans $\mathcal{I}$, c'est à dire dans $\mathcal{A}_0$. $\mathcal{A}_1$ étant un attracteur pour $\mathcal{A}_0$, $\mathcal{E}$ contient une configuration $C_1$ de $\mathcal{A}_1$.

De même, $\mathcal{A}_2$ étant un attracteur pour $\mathcal{A}_1$, $\mathcal{E}$ contient une configuration $C_2$ de $\mathcal{A}_2$. De proche en proche, on montre que $\mathcal{E}$ est nécessairement de la forme $(C_0,...,C_1,...,C_2,...,...,C_j)$ avec $C_i \in \mathcal{A}_i$.

$C_j$ appartenant a $\mathcal{A}_j$, c'est donc une configuration légitime. □

### C) Exécutions équivalentes

La preuve de la convergence tout comme l'évaluation de la complexité en temps se fait en considérant l'ensemble des exécutions possibles. La tâche est souvent vaste. Aussi la limite-t-on en restreignant le nombre des exécutions à considérer.

**Définition 2.3.7 : Exécutions équivalentes**
Soit $\mathcal{S}$ un système réparti, $\mathcal{E}$ et $\mathcal{E}'$ deux exécutions du système. $\mathcal{E}$ et $\mathcal{E}'$ sont *C-équivalentes* (ou encore *Equivalente en Convergence*) si la longueur de leur phase de stabilisation est identique. Par convention, deux exécutions ne convergeant pas (c'est à dire ayant des phases de stabilisation infinie) sont C-équivalentes.

On note $\mathcal{E} \overset{C}{\equiv} \mathcal{E}'$.

$\mathcal{E}$ est *C-equivalente* à $\mathcal{E}' + x$ si la phase de stabilisation de $\mathcal{E}'$ comporte $x$ transitions de plus que celle de $\mathcal{E}$.

On note $\mathcal{E} \overset{C}{\equiv} \mathcal{E}' + x$

Un exemple d'exécutions *C-équivalentes* est donné figure 2.6. Les trois exécutions ont $L_6$ pour première configuration légitime. $\mathcal{E}$ et $\mathcal{E}'$ ont un temps de stabilisation de 6. On a donc $\mathcal{E} \overset{C}{\equiv} \mathcal{E}'$. L'exécution $\mathcal{E}''$ ayant un temps de convergence de 4, on a $\mathcal{E} \overset{C}{\equiv} \mathcal{E}'' + 2$.

Supposons qu'il existe un ensemble d'exécutions $\mathcal{X}$ dont la convergence est facile à prouver. Alors si on montre que toute exécution est équivalente à une exécution de $\mathcal{X}$, la convergence du système est assurée.

**Lemme 2.3.8**
La relation *C-équivalente* est une relation d'équivalence.

**Preuve :** Une relation d'équivalence doit être réflexive, symétrique et transitive.

1. **Réflexive :** $\mathcal{E} \overset{C}{\equiv} \mathcal{E}$ car $\mathcal{E}$ a la même phase de stabilisation que $\mathcal{E}$ (!).
2. **Symétrique :** si $\mathcal{E} \overset{C}{\equiv} \mathcal{E}'$, les deux exécutions ont des phases de stabilisation de même longueur. On a donc aussi $\mathcal{E}' \overset{C}{\equiv} \mathcal{E}$.
3. **Transitive :** si $\mathcal{E} \overset{C}{\equiv} \mathcal{E}'$ et $\mathcal{E}' \overset{C}{\equiv} \mathcal{E}''$, les trois exécutions ont des phases de stabilisation de même longueur. On a donc aussi $\mathcal{E} \overset{C}{\equiv} \mathcal{E}''$.

De la même manière, on a $\mathcal{E} \overset{C}{\equiv} \mathcal{E}' + x \iff \mathcal{E}' + x \overset{C}{\equiv} \mathcal{E}$ et $\mathcal{E} \overset{C}{\equiv} \mathcal{E}' + x \wedge \mathcal{E}' \overset{C}{\equiv} \mathcal{E}'' + y \implies \mathcal{E} \overset{C}{\equiv} \mathcal{E}'' + x + y$ □

**Théorème 2.3.9**
Soit $\mathcal{S}$ un système. Si pour toute exécution de $\mathcal{S}$, il existe une exécution *C-équivalente* ayant un temps de convergence fini, alors le système converge.

**Preuve :** La convergence est assurée si toute exécution contient une configuration légitime. Or, une exécution quelconque est *C-équivalente* à une exécution ayant une phase de stabilisation finie. Elle a donc elle même une phase de stabilisation finie et contient donc une configuration légitime. □

La notion de *C-équivalence* est également grandement utilisé pour calculer la complexité en temps d'un algorithme.

**Théorème 2.3.10**
Soit $\mathcal{S}$ un système. Si pour toute exécution de $\mathcal{S}$, il existe une exécution *C-équivalente* ayant un temps de convergence en $O(M)$, alors le système se stabilise en $O(M)$.

**Preuve :** La phase de stabilisation est la pire des phases de stabilisation de toutes les exécutions possibles. Or, une exécution quelconque est *C-équivalente* à une exécution ayant une phase de stabilisation en $O(M)$. Elle a donc elle même une phase de stabilisation en $O(M)$. □

### 2.3.4 Complexité

L'étude de la complexité se divise en deux parties : la complexité en espace et la complexité en temps.

**La complexité en espace** est la taille des mémoires nécessaires à l'implémentation de l'algorithme. Pour l'évaluer, il suffit en général d'additionner la taille de toutes les variables utilisées par un processeur, puis de prendre le logarithme en base 2 du résultat obtenu (car un nombre de taille $n$ se code en utilisant $Log_2(n)$ bits). Le résultat final est la complexité en taille de l'algorithme.

Etablir la **complexité en temps** est souvent plus délicat. Il s'agit en effet de déterminer le temps mis par la pire des exécutions pour converger[4]. En général, on trouve une majoration du pire temps de convergence, puis on exhibe une exécution du système ayant un temps de convergence proche de la borne précédemment établie.

**Paramètres :** la complexité d'un algorithme est généralement paramétré en fonction de la taille $n$ du réseau considéré. Dans le cas d'algorithmes $k$-stabilisants, proportionnels et $k$-proportionnels rentrent également en ligne de compte le nombre de fautes tolérées par l'algorithme ainsi que le nombre effectif de fautes présentes dans la configuration initiale.

Classiquement, la lettre $k$ est utilisée pour noter le nombre maximum de fautes tolérées par un algorithme (généralement, $k$ est fixé à l'implémentation de l'algorithme) alors que $f$ dénote le nombre effectif de fautes frappant le réseau. En particulier, un algorithme $k$-proportionnel tolère jusqu'à $k$ fautes, mais assure une convergence polynomiale en $f$, le nombre de fautes frappant effectivement le réseau.

**Ordre de grandeur :** dans le cadre de l'auto-stabilisation, déterminer la complexité exacte d'un algorithme est une tâche à la foi délicate et ne présentant qu'un interêt moyen. Par exemple, si un algorithme a un temps de convergence de $2n$ transitions, un algorithme ayant un temps de convergence de $n$ transitions ne présente que peut d'interet puisque dans les deux cas, le temps de convergence reste proportionnel à la taille du réseau. Cela signifie que sur un réseau de taille giganstesque, aucun des deux algorithmes n'est implantable alors qu'ils le sont tous les deux sur des réseaux de petite

4. Dans cette thèse, nous ne considérons que la complexité au pire.

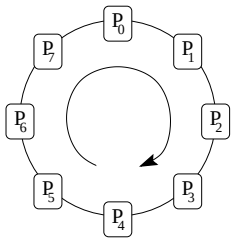



FIG. 2.7 – *Un anneau orienté*

taille. Aussi, dans les études de complexité, il est fait abstraction des détails pour ne se concentrer que sur les ordres de grandeur. Par exemple, les deux algorithmes dont nous venons de parler ont des complexités en espace de l'ordre de $n$ transitions. Les éventuelles constantes mutiplicatives ne sont pas prises en compte. Dans le cadre de la compléxité en espace, cela signifie entre autre chose que l'on confond le logarithme népérien et logarithme en base 2, les deux fonctions étant égales au facteur $\frac{1}{Log(2)}$ prêt.

Formellement, trois notations permettent d'exprimer les ordres de grandeurs : $O$ designe un majorant, $\Omega$ est un minorant alors que $\Theta$ est une approximation :

**Définition 2.3.11**
On dit d'un fonction $f$ qu'elle est d'un ordre de grandeur inferieur à $g$ (et on note $f = O(g)$) s'il existe une constante $C$ telle que $f(n)$ soit plus petit que $C.g(n)$

$$f(n) = O(g(n)) \Longleftrightarrow \exists C \text{ telle que } f(n) \leq C.g(n)$$

De même, $f$ est d'un ordre de grandeur supérieur à $g$ (on note $f = \Omega(g)$) s'il existe une constante $c$ telle que $f(n)$ soit plus grand que $c.g(n)$

$$f(n) = \Omega(g(n)) \Longleftrightarrow \exists c \text{ telle que } f(n) \geq c.g(n)$$

Enfin, $f$ est d'un ordre de grandeur comparable à $g$ (on note $f = \Theta(g)$) s'il existe deux constantes $c$ et $C$ telles que $f(n)$ soit plus petit que $C.g(n)$ et plus grand que $c.g(n)$.

$$f(n) = \Theta(g(n)) \Longleftrightarrow f(n) = O(g(n)) \text{ et } f(n) = \Omega(g(n))$$

## 2.4 Exclusion mutuelle sur un anneau

Tout au long de notre étude, nous allons présenter divers algorithmes d'exclusion mutuelle. Pour manipuler ces objets ont été développés un langage particulier et des définitions spécifiques, définitions que nous allons présenter ici.

### 2.4.1 Orientation d'un anneau

Dans un anneau, chaque processeur a deux voisins. Graphiquement, il est aisé de définir un sens de parcours de l'anneau, par exemple celui des aiguilles d'une montre (figure 2.7). Mais si l'anneau est anonyme, rien ne permet à un processeur de faire la différence entre ses deux voisins. Aussi fait-on généralement l'hypothèse que l'anneau

est orienté, c'est à dire que chaque processeur a un prédécesseur dont il est le successeur, et un successeur dont il est le prédécesseur.

**Définition 2.4.1 : Orientation d'un anneau**
Soit $G$ un graphe de taille $n$ ayant une structure d'anneau et dont tous les processeurs sont numérotés de $P_0$ à $P_{n-1}$. Le graphe est dit *orienté* si chaque processeur $P_i$ reconnait $P_{i-1}$ pour prédécesseur et $P_{i+1}$ pour successeur (avec $i \in [0..n]$ et $P_0 = P_n$).

$$Pred(P_i) = P_{i-1}$$
$$Succ(P_i) = P_{i+1}$$

Un exemple d'anneau orienté dans le sens indirect est donné figure 2.7. $P_1$ est le successeur de $P_0$ et le prédécesseur de $P_2$.

**Notation :** par commodité, lorsque les processeurs ne sont pas numérotés, le prédécesseur de $P$ est également noté $P^-$ alors que son successeur est noté $P^+$. Cette notation se généralise à un groupe de plusieurs processeurs. Par exemple, $(((P^+)^+)^+)^+$ est noté $P^{4+}$.

### 2.4.2 Exclusion mutuelle

Le problème de l'exclusion mutuelle se pose quand plusieurs utilisateurs veulent utiliser simultanément une même ressource ne pouvant répondre qu'à une seule demande. On a alors besoin de disposer d'un protocole leur interdisant d'accéder en même temps à l'unique ressource tout en leur garantissant tout de même un accès éventuellement différé.

Formellement, le problème de l'exclusion mutuelle est défini de la manière suivante :

**Définition 2.4.2 : Exclusion mutuelle**
Un système réparti vérifie la spécification de *l'exclusion mutuelle* si chaque processeur du réseau dispose d'un prédicat booléen vérifiant :

1. Dans chaque configuration, un unique processeur a son prédicat à vrai.
2. Au cours de chaque exécution, chaque processeur a son prédicat à vrai infiniment souvent.

On dit d'un processeur ayant son prédicat à vrai qu'il a un *privilège*.

### 2.4.3 Interprétation des privilèges

De nombreux algorithmes d'exclusion mutuelle définissent la notion de privilège relativement aux règles gardées des processeurs. Un processeur $P$ est privilégié s'il peut appliquer une certaine règle, c'est à dire si la condition booléenne de la règle est vraie. En général, le fait d'exécuter effectivement la règle en question lui fait modifier les valeurs de ses variables propres. Il perd ainsi son privilège. Dans le modèle à état, les variables de $P$ peuvent être lues par les voisins de $P$. Il est donc possible que l'action de $P$ ait également pour effet de faire apparaître un privilège chez un de ses voisins. Au final, $P$

a perdu son privilège et un privilège vient d'apparaître chez un de ses voisins. Tout se passe comme si le processeur $P$ avait "transmis" son privilège à un de ses voisins.

Dans ce cas, on peut alors considérer un algorithme d'exclusion mutuelle comme un ensemble de règles permettant de réguler la circulation (et éventuellement les collisions) de privilèges sur un graphe. Le privilège est alors considéré comme une entité propre pouvant même avoir des caractéristiques spécifiques[5]. D'où l'utilisation d'une terminologie spécifique à cette approche.

**Définition 2.4.3 : mouvement de privilège**
Un *processeur privilégié* est le possesseur d'un privilège. Dans le modèle à état, quand un processeur privilégié applique une règle qui lui fait perdre son privilège et en fait apparaitre chez un de ses voisins, on dit qu'il *transmet* son privilège (à un de ses voisins). Si $P$ vient de transmettre son privilége[6] $J$ à $P'$, on dit que $J$ vient de *franchir* $P$ et qu'il *arrive* en $P'$.
Si $P$ ne transmet pas son privilège $J$, on dit que $J$ est *immobile*.

**Définition 2.4.4 : Support**
Le processeur $P$ possédant un privilège $J$ est appelé *support* de $J$.

$$P = Support(J).$$

Grâce à la notion de support, toutes les définitions propres aux processeurs peuvent être étendues aux privilèges. Par exemple, la distance entre un privilège $J$ et un processeur $P$ est la distance entre le support de $J$ et $P$. De même, la distance entre deux privilèges est la distance entre leur support respectif.

Quand un processeur $P$ applique une règle lui faisant perdre son privilège et qu'un de ses voisins $P'$ possède déjà un privilège, deux cas sont possibles :

1. Si un privilège apparîit chez $P''$, un autre voisin de $P$, alors $P$ a transmis son privilège a $P''$.
2. Si aucun privilège n'apparaît parmi les voisins de $P$, on considère que $P$ a transmis son privilège a $P'$. Celui-ci ne pouvant être deux fois privilégié, on dit que les deux privilèges (celui qui était en $P$ et celui qui était en $P'$) fusionnent.

**Définition 2.4.5 : Fusionner**
Soit $J$ un privilège et $P$ son support. $J$ fusionne avec un autre privilège s'il est transmis à un processeur ayant un privilège immobile, ou s'il est immobile et qu'un privilège arrive en $P$.

5. Attention toutefois au fait que, contrairement à ce qui se passe dans le modèle à passage de message (où les privilèges sont vraiment des entités définies par le modèle), les privilèges du modèle à état ne doivent leur existence qu'aux valeurs de certains processeurs. Leur personnification constitue donc plus un abus de langage qu'une réalité physique.

6. Les privilèges sont également appelés "jetons". Aussi, lettre $P$ étant déjà utilisée pour désigner les processeurs, nous noterons $J$ les privilèges.

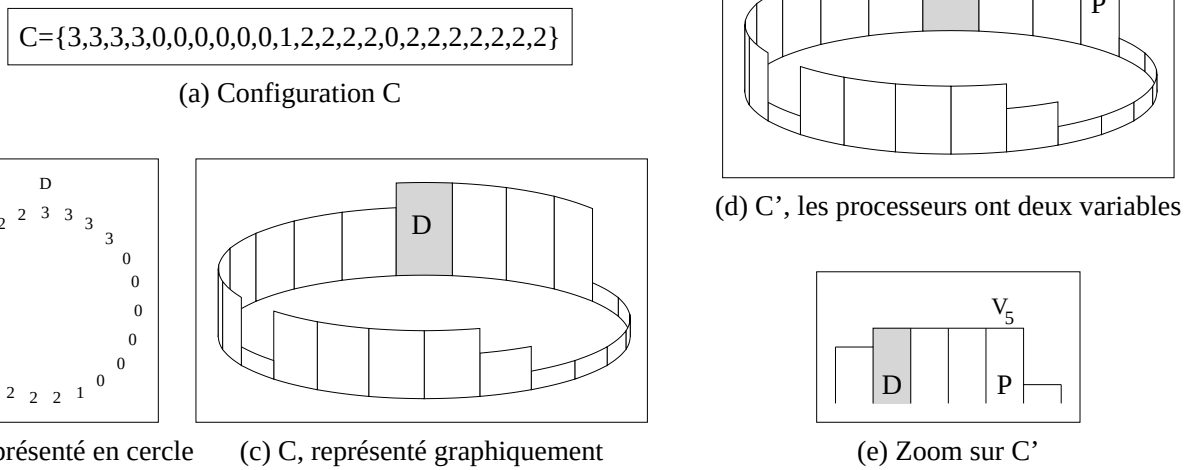



FIG. 2.8 – *Représentation graphique d'une configuration*

**Sur un anneau orienté,** dans les exemples que nous allons utiliser par la suite, les privilèges se déplacent tous dans le même sens. Bien souvent, il sera nécessaire de parler du plus proche prédécesseur de $P$ possédant un privilège. Ce processeur est appelé privilège prédécesseur.

**Définition 2.4.6 : Privilège Prédécesseur / Successeur / Voisins**
Le *privilége prédécesseur* d'un processeur $P$ (noté $PrivPred(P)$) est le plus proche prédécesseur de $P$ possédant un privilège.
De même, le *privilége successeur* de $P$ (noté $PrivSucc(P)$) est le plus proche successeur de $P$ possédant un privilège.
L'ensemble constitué du privilège prédécesseur et du privilège successeur de $P$ forme les privilèges voisins de $P$.

Enfin, lorsqu'un privilège se déplace, il modifie la valeur des processeurs par lesquels il passe, propageant ainsi la valeur du prédécesseur de son support.

**Définition 2.4.7 : Valeur d'un privilège**
Soit $J$ un privilège et $P$ son support. La valeur du privilège $J$ est la valeur que prendra $P$ lorsqu'il aura transmis $J$.

$$Valeur(J) = Priv(P^-)$$

### 2.4.4 Interprétation graphique

Tout au long de cette thèse, nous illustrerons nos propos par des exemples. Dans le cas d'un algorithme où chaque processeur est muni d'une unique variable, une configuration est l'ensemble des valeurs de tous les processeurs. Par exemple, (figure2.8.(a))

$C$ = (3,3,3,3,0,0,0,0,0,0,1,2,2,2,2,2,0,2,2,2,2,2,2) est une configuration d'un anneau semi-uniforme de taille 23 (la valeur du processeur distingué est la première valeur du vecteur). Une autre manière de représenter cette même configuration est de la dessiner sous forme d'anneau (figure2.8.(b)) en plaçant le distingué en haut de l'anneau. Graphiquement, on peut également dessiner une tour où chaque processeur est symbolisé par une portion de mur plus ou moins haute selon sa valeur (figure 2.8.(c)). Le distingué est alors reprénsenté en gris.

Dans un cas plus compliqué où les processeurs ont plusieurs variables, on peut représenter les configurations en omettant certaines variables, ou encore inscrire la valeur de la variable au dessus du processeur correspondant. Par exemple, dans la configuration $C'$ de la figure 2.8.(d), le processeur $P$ a une de ses variables à 4 (hauteur du mur), l'autre à $V_5$.

Au final, lorsqu'une exécution ne fait agir que quelques processeurs, il pourra être intéressant de ne considérer qu'une portion de la configuration totale. Dans ce cas, seul un pan de mur sera représenté (figure 2.8.(e)).

# Première partie

# Le Cas Asynchrone

*ou*

# TEST : un oracle plus vrai que nature

# Chapitre 3

# Exclusion mutuelle de Dijkstra

En 1974, Dijkstra présente un algorithme auto-stabilisant d'exclusion mutuelle sur un anneau. A travers son article, il définit un certain nombre de concepts (comme la notion de privilège). Dans différents chapitres de cette thèse, plusieurs algorithmes d'exclusion mutuelle sont présentés. Certains d'entre eux, assez complexes, reposent sur les mêmes bases que celui de Dijkstra. Aussi nous a-t-il paru interessant de faire une présentation détaillé de cet algorihtme fondateur, tant au point de vue de la modularité (pour introduire progressivement les concepts utilisés par les chapitres à suivre) que pour pouvoir établir des comparaisons de complexité.

Dans ce chapitre, nous présentons donc une étude complète de l'algorithme de Dijkstra. Après avoir exposé ses principes fondamentaux et les hypothèses nécessaires à son bon fonctionnement, nous présentons son pseudo code avant de donner la preuve qu'il est auto-stabilisant. Enfin, nous étudions sa complexité, aussi bien dans le cadre général de l'auto-stabilisation que dans celui plus restreint de la $k$-stabilisation.

## 3.1 Principes généraux

Lorsque l'on considère le problème de l'exclusion mutuelle auto-stabilisante sur un anneau, on se retrouve principalement confronté à deux difficultés.

La première est d'assurer **l'existence d'un privilège**. En effet, si une configuration sans privilège est permise, l'ensemble des processeurs peut rester inactif en pensant qu'un privilège existe quelque part sur l'anneau et qu'il suffit d'attendre pour le recevoir. Remédier à cela peut être coûteux ou entraîner des hypothèses supplémentaires sur le réseau[1]. Dijkstra évite le problème en définissant les privilèges de manière à ce que leur existence soit assurée dans toutes les configurations possibles.

Le deuxième problème est d'assurer une disparition progressive des **privilèges surnuméraires**. Classiquement, si deux privilèges sont présents dans une configuration, toute exécution doit invariablement conduire vers une configuration où seul un des deux privilèges subsiste. Informellement, Dijkstra permet cela en donnant à un des processeurs de l'anneau un rôle de "filtre à privilèges surnuméraires".

1. La plus commune étant l'introduction d'un démon probabiliste, comme dans [?]

## 3.2 Hypothèses & résultats

Dijkstra se place dans le cadre d'un **modèle à état** (chaque processeur peut lire les variables de ses voisins, lire et écrire dans ses variables propres). Il considére un **anneau semi-uniforme** (un seul processeur, appelé le processeur *distingué* et noté $D$, peut avoir un code different des autres) **unidirectionnel orienté** (l'anneau est orienté -les notions de successeurs et de prédécesseurs sont définies- et chaque processeur ne peut lire que ses variables propres et celles de son prédécesseur). L'exécution de l'algorithme est régie par un **démon centralisé**[2] (à chaque transition, un seul processeur peut exécuter sa règle gardée). Sous ces hypothèses, Dijkstra propose un algorithme d'exclusion mutuelle auto-stabilisant ayant un temps de stabilisation quadratique en la taille de l'anneau :

**Théorème 3.2.1**
L'algorithme de Dijkstra présenté figure 3.1 est auto-stabilisant pour le problème de l'exclusion mutuelle. De plus, si la taille de l'anneau est $n$, sa mise en application nécessite $O(log(n))$ bits par processeur et son temps de stabilisation (complexité au pire) est de $O(n^2)$ transtions.

## 3.3 Algorithme

L'algorithme de Dijkstra est présenté figure 3.1.

**Prédicat : Définition d'un Privilège**

$$Privilege(P) \quad \Leftrightarrow \begin{cases} Priv(P) = Priv(P^-) & \text{si } P \text{ est le processeur distingué.} \\ Priv(P) \neq Priv(P^-) & \text{si } P \text{ est un processeur quelconque.} \end{cases}$$

**Action : Transmettre un privilège**

$$Transmettre(P) \quad \Leftrightarrow Priv(P) \longleftarrow \begin{cases} Priv(P^-) + 1 & \text{Si } P \text{ est le distingué.} \\ Priv(P^-) & \text{Sinon.} \end{cases}$$

**Règle gardée**

$$R \quad Privilege(P) \quad \Longrightarrow Transmettre(P)$$

FIG. 3.1 – *Premier algorithme auto-stabilisant de Dijkstra*

Chaque processeur $P$ dispose d'une unique variable $Priv(P)$ à valeur dans $[0..n-1]$. Les opérations d'incrémentations de cette variable se font modulo $n$.

Elle permet de définir le prédicat **Privilege**. Pour le processeur distingué $D$, ce prédicat est vrai lorsque la valeur de sa variable $Priv(D)$ est la même que celle de son prédécesseur $Priv(D^-)$ Pour un processeur quelconque $P$, il est vrai lorsque la valeur de $Priv(P)$ est différente de celle son prédécesseur $Priv(P^-)$

La seule action possible pour un processeur est de **transmettre** son privilège à son successeur. Pour cela, il modifie sa variable $Priv(P)$ de manière à ce qu'il n'ait plus de

2. L'algorithme est également auto-stabilisant sous un démon réparti, mais nous n'en donnons pas la preuve ici.

privilège.

L'algorithme se compose d'une unique règle **R** : lorsqu'un processeur est privilègié, il transmet son privilège.

## 3.4 Exemples d'exécutions

Divers exemples d'exécution sont présentés figure 3.2. La première (3.2.(b)) part d'une configuration légitime $L_0$. Seul le processeur $P_0$, le distingué, possède un privilège. Il le transmet à $P_1$ (configuration $L_1$), qui à son tour le transmet à $P_2$, ainsi de suite. Après 7 transitions, le privilège a fait un tour complet d'anneau (configuration $L_7$). Il est de nouveau en possession de $P_0$, prêt à repartir pour un nouveau tour.

La figure 3.2(c) présente toutes les exécutions possibles à partir de la configuration illégitime $C_0$. Initialement, les trois processeurs $P_3$, $P_4$ ou $P_7$ ont des privilèges. Selon que le démon permet à l'un ou l'autre d'agir, la deuxième configuration de l'exécution sera $C_1$, $C_1'$, ou $C_1''$. Sur cet exemple particulier, on constate que quelle que soit l'exécution choisie par le démon, elle contient toujours une configuration dans laquelle un seul privilège est présent (configurations cerclées sur la figure).

## 3.5 Preuve

Une preuve de cet algorithme peut-être trouvée dans [Tel94] (avec une preuve que le temps de convergence est de $O(n^3)$ transitions).

Nous donnons ici une preuve de l'algorithme légèrement différente qui nous permet d'obtenir une complexité en temps de $O(n^2)$ transitions.

La preuve de l'auto-stabilisation se fait en trois étapes :

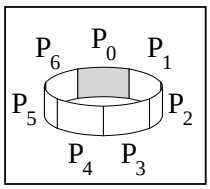
### 3.5.1 Choix des configurations légitimes

L'ensemble des configurations légitimes choisies est l'ensemble des configurations dans lesquelles un seul processeur est privilègié.

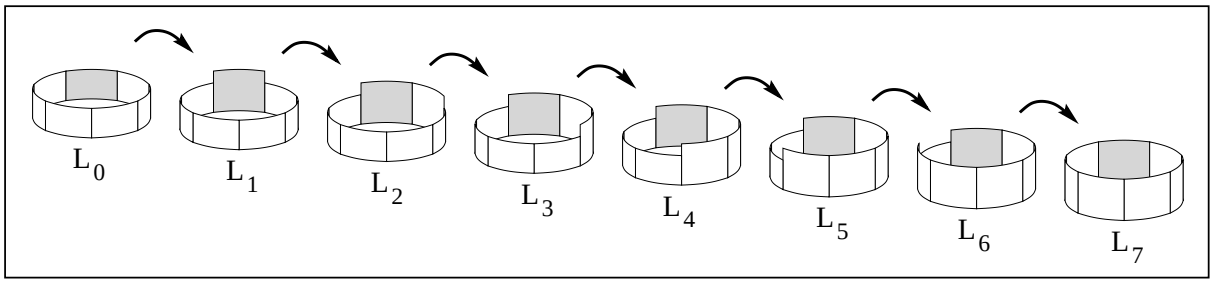
**Définition 3.5.1**
Une configuration $L$ est légitime si et seulement si il existe un unique processeur dans $L$ possédant un privilège.

### 3.5.2 Correction

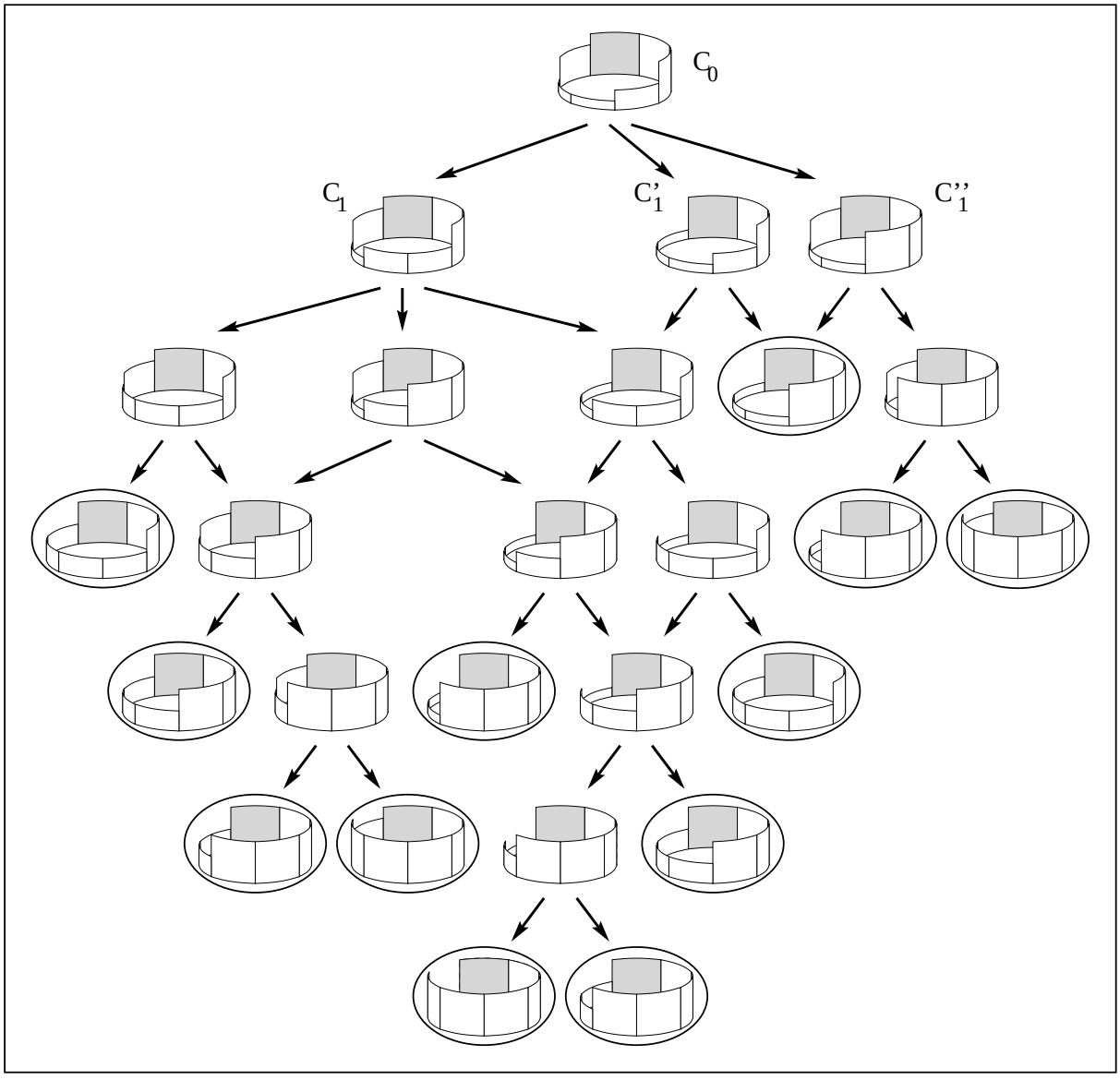
**Survol de la preuve :** prouver la correction, c'est établir que toute exécution ayant une configuration initiale légitime vérifie la spécification de l'exclusion mutuelle. Pour cela, nous montrons que dans toute configuration légitime il existe au moins un privilège et que le nombre de privilèges n'augmente jamais au cours d'une exécution. Ces deux propriétés nous assurent que toutes les configurations d'une exécution légitime




(a) Numérotation des processeurs




(b) Exécution partant d'une configuration légitime




(c) Exécutions possibles à partir de la configuration illégitime $C_0$

FIG. 3.2 – *Exemple d'exécution de l'algorithme de Dijkstra*

contiennent exactement un privilège. Par ailleurs, nous montrons que toutes les exécutions sont infinies (il n'existe pas de configuration terminale). Le privilège est donc transmis une infinité de fois. Comme il circule toujours dans le même sens, tous les processeurs le recevront infiniment souvent, ce qui établit la correction.

**Lemme 3.5.2**
Dans toute configuration, il existe au moins un processeur privilégié.

**Preuve :** Soit $C$ une configuration du système. Un processeur est privilégié quand il peut appliquer sa règle. Montrons qu'il existe toujours un tel processeur :

1. Si le processeur distingué a la même valeur que son prédécesseur, il peut appliquer sa règle.
2. Si le processeur distingué n'a pas la même valeur que son prédécesseur, il existe au moins un autre processeur dont la valeur est différente de celle de son prédécesseur. Ce processeur peut appliquer sa règle.

Dans tous les cas, un processeur privilégié existe. □

**Remarque 3.5.3**
Pour un processeur donné $P$, la présence ou l'absence de privilège dépend des deux variables $Priv(P)$ et $Priv(P^-)$. Donc, si $P$ modifie sa valeur, les seuls processeurs pouvant subir des apparitions ou disparitions de privilèges sont $P$ et $P^+$.

**Lemme 3.5.4**
Au cours d'une exécution, le nombre de privilège n'augmente jamais.

**Preuve :** Soit $C_1 \longrightarrow C_2$ une transition du système. Le démon étant centralisé, entre $C_1$ et $C_2$, un unique processeur agit.

Seul un processeur $P$ ayant un privilège dans $C_1$ peut appliquer sa règle. Ce faisant, il perd son privilège. Or seuls $P$ et $P^+$ peuvent subir des apparitions ou disparitions de privilège (remarque 3.5.3). D'ou, même si $P^+$ n'a pas de privilège dans $C_1$ et en a un dans $C_2$, le nombre global de privilèges de $C_2$ ne peut pas être plus important que celui de $C_1$. □

**Lemme 3.5.5**
Il n'existe pas de configuration terminale.

**Preuve :** Dans toute configuration, il existe au moins un processeur privilègié (lemme 3.5.2) qui, par définition du privilège, peut appliquer sa règle gardée. □

**Remarque 3.5.6**
Dans une configuration légitime, un processeur ne peut transmettre son privilège qu'à son successeur. En effet, un unique processeur $P$ est privilégié. $P^+$ n'a donc pas de privilège. Lorsque $P$ applique sa règle, il modifie sa valeur et perd son privilège. Ce faisant, il donne un privilège à $P^+$.

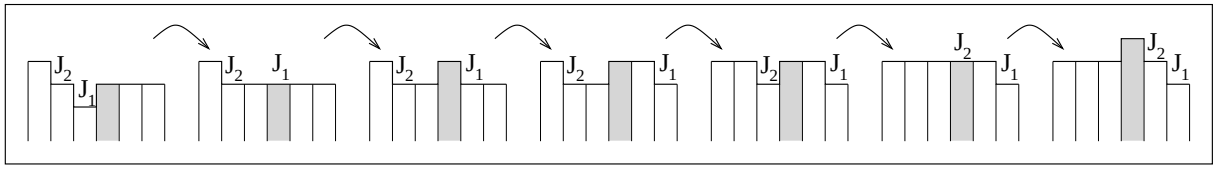



FIG. 3.3 – *Illustration du lemme 3.5.8*

Nous pouvons maintenant prouver la correction :

**Lemme 3.5.7**
Toute exécution légitime vérifie la spécification de l'exclusion mutuelle.

**Preuve :** Dans une configuration légitime, un seul processeur est privilégié (définition 3.5.1). Le nombre de privilège n'augmentant pas (lemme 3.5.4) et ne pouvant être nul (lemme 3.5.2), il reste constant et égal à 1 durant toute l'exécution. De plus, un processeur ne transmettant son privilège qu'à son successeur (remarque 3.5.6), tous les processeurs le recoivent infiniment souvent. □

### 3.5.3 Convergence & complexité en temps

**Survol de la preuve :** pour montrer la convergence, nous devons établir que toute exécution contient une configuration légitime. Pour cela, nous bornons le nombre de fois où un privilège peut franchir le processeur distingué tout en étant dans une configuration illégitime. L'existence de cette borne assure la convergence et permet d'évaluer la complexité en temps de l'algorithme.

**Lemme 3.5.8**
Un privilège ne peut franchir le processeur distingué $D$ plus d'une fois que si aucun autre privilège ne franchit $D$ entre ses deux passages.

**Preuve :** Supposons qu'un privilège $J_1$ franchisse $D$, puis qu'un autre privilège $J_2$ franchisse également $D$. Montrons que $J_1$ ne pourra plus franchir $D$ :

Soit $x$ la valeur de $D$ avant le passage de $J_1$. Après le passage de $J_1$ en $D$, $Priv(D)$ et $Valeur(J_1)$ ont pour valeur $x+1$. Si $J_2$ franchit $D$, $Priv(D)$ aura pour valeur $x+2$. $J_1$ ne peut franchir $D$ que si sa valeur est la même que celle de $D$. Donc $J_1$ ne peut franchir $D$ une deuxième fois que si $D$ retrouve la valeur $x+1$. Or $D$ ne peut incrémenter sa variable que de 1, et cela nécessite à chaque fois le passage d'un privilège. Pour que $D$ prenne à nouveau la valeur $x+1$, il faudrait que $n-1$ nouveaux privilèges franchissent $D$, ce qui est impossible car au plus $n-1$ privilèges (en incluant $J_1$ et $J_2$) peuvent être présents simultanement sur l'anneau. □

Un exemple illustrant cette preuve est présenté figure 3.3. Lorsque le privilège $J_1$ a franchi $D$ et que $J_2$ l'a franchi à son tour, la valeur de $J_1$ est trop faible pour pouvoir

franchir $D$ à nouveau.

**Corollaire 3.5.9**
Une configuration dans laquelle $D$ reçoit un même privilège pour la seconde fois est légitime.

**Preuve :** Si un privilège $J_1$ franchit $D$ deux fois, c'est qu'aucun autre privilège n'a franchi $D$ entre les deux passages de $J_1$ (lemme 3.5.8). D'où $J_1$ est seul sur l'anneau et la configuration est légitime. □

**Lemme 3.5.10**
Soit $\mathcal{E}$ une exécution du système et $C_0$ sa configuration initiale. Alors l'algorithme converge vers une configuration légitime en au plus $O(n^2)$ transitions.

**Preuve :** $C_0$ compte au plus $n$ privilèges. Chacun de ces privilèges peut avancer d'au plus $2n-1$ pas avant de franchir $D$ une deuxième fois. Comme il n'existe pas de configuration terminale (lemme 3.5.5), après moins de $2n^2$ transitions, un privilège franchi $D$ pour la seconde fois. L'exécution est alors dans une configuration légitime (corollaire 3.5.9). □

### 3.5.4 Bilan

Tous les éléments nécessaires à la preuve du théorème 3.2.1 sont maintenant réunis.

**Théorème 3.5.1**
L'algorithme de Dijkstra (figure 3.1, page 60) est auto-stabilisant pour le problème de l'exclusion mutuelle. De plus, si la taille de l'anneau est $n$, sa mise en application nécessite $O(log(n))$ bits par processeur et son temps de stabilisation est de $O(n^2)$ transitions.

**Preuve :** L'ensemble de configurations légitimes $\mathcal{L}$ (définie au 3.5.1) verifie la correction (lemme 3.5.7) et la convergence avec une phase de stabilisation de l'ordre de $n^2$ étapes (lemme 3.5.10). De plus, l'algorithme utilise une unique variable de taille $n$. D'où le système est auto-stabilisant pour l'exclusion mutuelle, son temps de convergence est $O(n^2)$ transition et il requiert $O(log(n))$ bits par processeur. □

## 3.6 Etude d'un faible nombre de fautes chez Dijkstra

L'algorithme de Dijkstra est auto-stabilisant. Sa phase de stabilisation est de l'ordre de $n^2$. Cela signifie qu'à partir d'une configuration initiale quelconque, une exécution converge vers une configuration légitime en un temps proportionnel à $n^2$.

Ce résultat est valable pour des corruptions totales. Il peut sembler légitime de se demander ce qu'il advient quand la configuration initale ne comporte qu'un petit nombre de fautes. On serait en particulier en droit d'espérer qu'une faible corruption entraîne une rapide convergence.

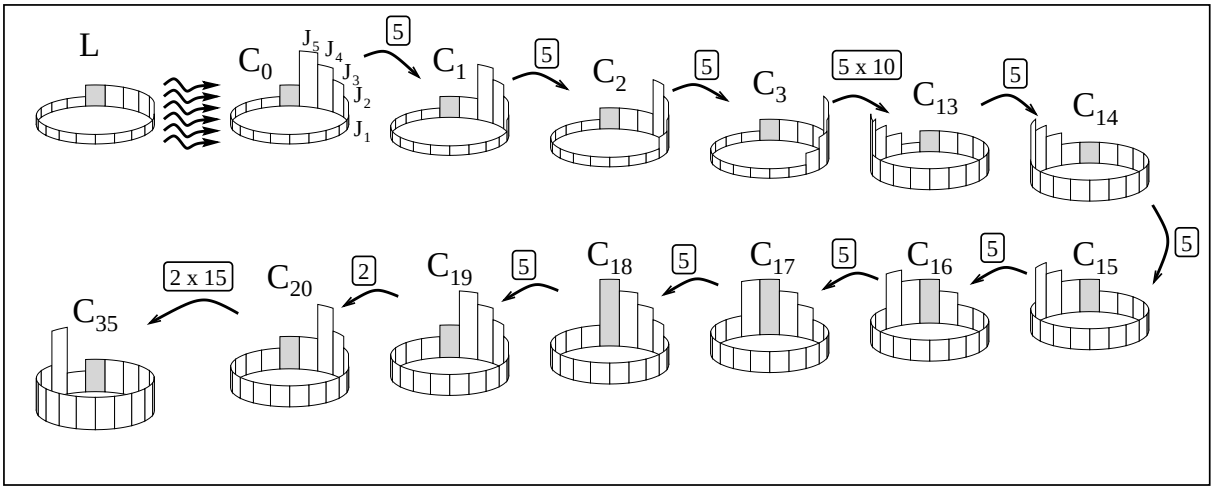



FIG. 3.4 – *Exemple d'exécution longue*

Malheureusement, même lorsque l'on considère des configurations initiales faiblement corrompues, les performances de l'algorithme ne sont que peu améliorées. En particuler, on peut trouver une exécution ayant initialement un unique privilège et ayant une phase de stabilisation de l'ordre de $n$. De manière plus générale, la phase de stabilisation est proportionnelle au nombre initial de fautes mais aussi à la taille de l'anneau.

**Lemme 3.6.1**
Si, dans l'algorithme de Dijkstra, le nombre de processeurs corrompus présents dans la configuration initiale est $f$, l'algorithme a un temps de stabilisation de $O(nf)$ transitions.

**Preuve :** $C_0$ compte $f$ processeurs corrompus. Chaque corruption peut entraîner l'apparition de 2 privilèges. Il y a donc au maximum $2f + 1$ privilèges dans $C_0$. Chacun de ces privilèges peut avancer d'au plus $2n - 1$ pas avant de franchir $D$ une deuxième fois. Comme il n'existe pas de configuration terminale (lemme 3.5.5), après moins de $(2f+1)\times(2n-1)$ transitions, un privilège franchi $D$ pour la seconde fois. L'exécution est alors dans une configuration légitime (corollaire 3.5.9). □

$5nf$ est un majorant du temps de stabilisation. Malheureusement, il est possible de construire une exécution dont le temps de convergence est proportionnel à cette borne :

**Lemme 3.6.2**
Il existe une exécution convergeant en plus de $\Omega(nf)$ transitions.

**Preuve :** (Cette preuve est illustrée par un exemple figure 3.4) Soit un anneau. On numérote tous ses processeurs dans le sens indirect en commençant par $D$ : $D = P_0$, $D^+ = P_1$, ..., $P_{n-1} = D^-$. Soit alors $C_0$ la configuration définie de la manière suivante :

1. Pour le processeur distingué, on pose $Val(D) = 2$.
2. Pour tout $i$ dans $[1..f + 1]$, on pose $Val(P_i) = f - i + 3$.

3. Pour tous les autres processeurs, $Val(P) = 1$

Cette configuration est bien dans $\mathcal{B}_f(L)$. En effet, si $L$ est la configuration légitime dans laquelle tous les processeurs de $P_0$ à $P_{f+1}$ sont a 2 et tous les autres sont à 1, la distance entre $C_0$ et $L$ est $f$.

Pour plus de commodité, numérotons les privilèges dans le sens direct. Considérons alors l'exécution $\mathcal{E}$ où le démon permet d'abord à $J_1$ d'agir, puis à $J_2$ et ainsi de suite jusqu'à $J_{f+2}$. Après $f+1$ transitions, l'exécution atteint la configuration $C_1$ dans laquelle tous les privilèges ont avancé d'un processeur. Si le démon réitére son choix, tous les privilèges vont à nouveau avancer d'un processeur. Après $2f+2$ transitions, l'exécution atteint la configuration $C_2$ dans laquelle tous les privilèges ont avancé de deux processeurs. Ainsi de suite.

Le privilège $J_1$ à pour valeur 2. Il peut donc franchir $D$. Après son passage, $D$ a pour valeur 3 et $J_2$ peut aussi franchir $D$. De même, tous les processeurs vont pouvoir franchir $D$. Après $nf+f$ transitions, l'exécution atteint donc $C_n$, la configuration dans laquelle tous les privilèges ont fait exactement un tour d'anneau. A partir de $C_n$, si le démon n'active plus que les deux derniers priviléges à avoir franchi $D$, ils peuvent tous les deux parcourir l'anneau dans sa quasi-intégralité, soit $2n-2$ transitions supplémentaires avant de parvenir à $C_{n+1}$ qui n'est pas une configuration légitime. La phase de stabilisation de $\mathcal{E}$ est donc supérieure à $nf+f+2n$. □

Un exemple de ce genre d'exécution est donnée figure 3.4. L'anneau considéré est de taille 19. $C_0$ est obtenue en corrompant trois processeurs de la configuration légitime $L$. Après 5 transitions où le démon aura fait avancer successivement $J_1$, $J_2$, $J_3$, $J_4$ et enfin $J_5$, l'exécution sera dans la configuration $C_1$. En itérant, après $5 \times 19$ transitions, la configuration $C_{19}$ sera atteinte. Si le démon fait alors avancer uniquement $J_4$ et $J_5$, l'exécution arrivera dans la configuration $C_{35}$ après $2 \times 16$ transitions supplémentaires. Au total, près de $5 \times 19$ transitions séparent $C_0$ de $C_{35}$.

Informellement, ce résultat signifie qu'une faute relativement bénigne peut perturber l'exécution pendant longtemps et qu'une simple corruption peut faire un tour complet de l'anneau avant la stabilisation.

# Chapitre 4

# Impossiblité de l'Exclusion mutuelle répartie proportionnelle

Comme la plupart des algorithmes auto-stabilisants, celui de Dijkstra a un temps de convergence dépendant de la taille du réseau. Même quand la corruption initiale est de faible importance, la phase de stabilisation reste longue. Il serait être intéressant de construire un algorithme d'exclusion mutuelle tel qu'une faible corruption entraîne une rapide convergence.

Hélas, la réalité des choses est loin d'être aussi complaisante et le long temps de convergence de Dijkstra peut se généraliser : il n'existe pas d'algorithme d'exclusion mutuelle proportionnel sous un démon centralisé.

## 4.1 Informellement...

Ce résultat d'impossibilité s'explique par le fait que dans une configuration donnée, le démon a toute latitude sur le choix du prochain processeur à activer. Aussi, en cas de faible corruption, il peut négliger les processeurs corrompus et leurs voisins pour ne donner la possibilité d'agir qu'au processeur ayant le privilège présent sur l'anneau avant la corruption. Ce privilège peut alors faire un tour presque complet de l'anneau avant d'interférer avec la zone corrompue. En fait, tout se passe comme si le démon choisissait délibérement d'ignorer la corruption et de faire fonctionner l'algorithme comme si rien ne s'était passé. Ainsi, en ne les activant jamais, il ôte aux corrompus l'opportunité de retrouver une valeur correcte rapidement.

L'exemple donné figure 4.1 illustre cet état des choses. La configuration $C$ est ob-

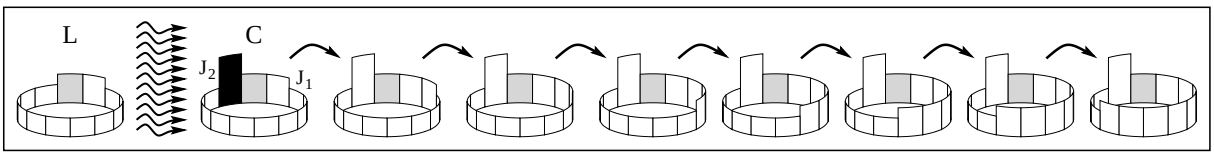



FIG. 4.1 – *Stabilisation proportionnelle impossible*

tenue à partir de la configuration légitime $L$ en corrompant le processeur $D^-$. Deux privilèges existent dans $C$. $J_1$ était déjà présent dans $L$ alors que $J_2$ doit son existence à la corruption. Si le démon choisit de ne pas faire avancer $J_2$, , $J_1$ va pouvoir parcourir tout l'anneau avant qu'une configuration légitime soit atteinte.

## 4.2 Hypothèses & résultats

Toujours dans le **modèle à état**, nous considérons un **anneau orienté** régi par un **démon centralisé**. Cette fois-ci, l'anneau est **bidirectionnel** et **non-anonyme**.

Sous ces hypothèses, un algorithme d'exclusion mutuelle ne peut pas être proportionnel.

**Théorème 4.2.1**
Sous les hypothèses précédentes, il n'existe pas d'algorithme d'exclusion mutuelle ayant un temps de convergence inférieur à $n-2$ transitions.

## 4.3 Preuve

**Considérations générales :** Nous considérons ici un système réparti $\mathcal{S}$ auto-stabilisant pour l'exclusion mutuelle. On suppose que chaque processeur $P$ est doté d'un ensemble de variables dont le contenu est désigné par $Variable(P)$. Si $C_1$ et $C_2$ sont deux configurations, l'égalité $Variable_{C_1}(P) = Variable_{C_2}(P)$ signifie que les variables de $P$ ont la même valeur dans les configurations $C_1$ et $C_2$. De même, deux configurations sont identiques si toutes les variables de tous les processeurs ont mêmes valeur dans l'une et dans l'autre des configurations.

**Survol de la preuve** Un démon centralisé fait agir les processeurs les uns après les autres. Aussi, dans toute exécution, certains processeurs peuvent n'être activés que longtemps (c'est à dire de l'ordre de $n$ transitions) après le début de l'exécution. Si ces processeurs-là étaient justement des corrompus, le temps de stabilisation est de l'ordre de $n$.

La preuve du théorème est basée sur le fait que lorsque deux processeurs d'un anneau sont distants, l'ordre dans lequel ils agissent n'a pas d'importance sur le déroulement général de l'exécution. Fort de ce principe, nous montrons que pour tout algorithme d'exclusion mutuelle, il est possible de construire une exécution dont trois processeurs contigus sont ignorés par le démon pendant les $n-3$ premières transitions, et cela même si le processeur encadré par les deux autres est corrompu. Cela établit le résultat d'impossibilité.

**Note sur les illustrations :** pour rester le plus général possible et ne pas nous restreindre aux seuls algorithmes définissant la présence d'un privilège relativement à une variable, les illustrations présentées dans cette section ne seront plus faites sous forme

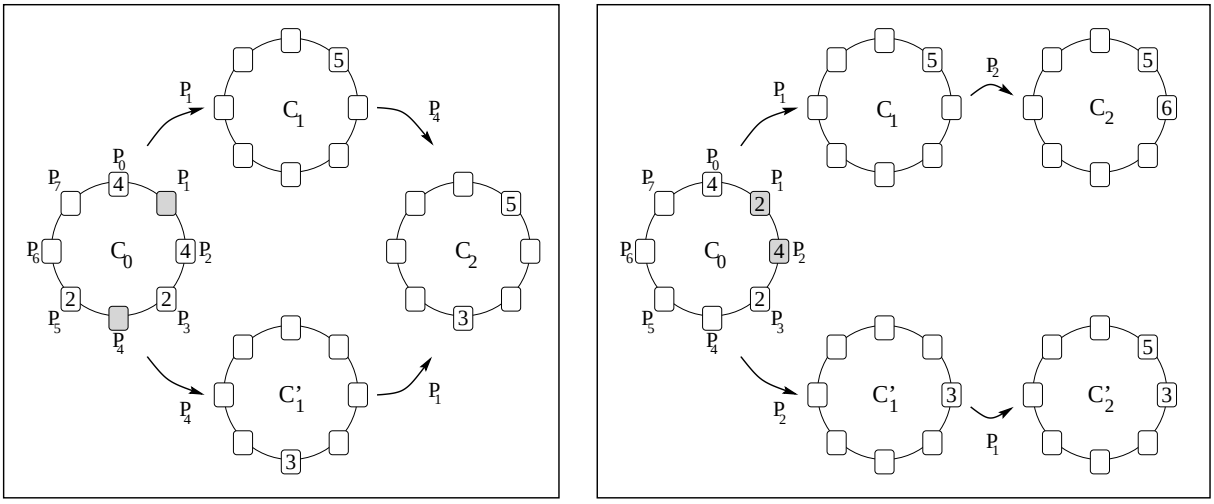



FIG. 4.2 – *Indépendance de processeurs*

de tour mais simplement sous forme d'anneau avec des processeurs (comme l'exemple d'orientation d'anneau, figure 2.7, page 51).

**Définition 4.3.1**
Soit $C_0$ une configuration, $P_1$ et $P_2$ deux processeurs activables. $P_1$ et $P_2$ sont indépendants si l'ordre dans lequel le démon les fait agir n'a pas d'influence sur le déroulement de l'exécution. On note alors $P_1 \parallel P_2$.

$$P_1 \parallel P_2 \iff \begin{array}{ccccc} & \xrightarrow{P_1} & C_1 & \xrightarrow{P_2} & \\ C_0 & & & & C_2 \\ & \xrightarrow{P_2} & C'_1 & \xrightarrow{P_1} & \end{array}$$

Par convention, un processeurs n'est pas indépendant de lui-même.

Un exemple illustrant cette notion est présenté figure 4.2. Dans la configuration $C_0$, $P_1$ et $P_4$ peuvent agir. On suppose dans cet exemple que l'action d'un processeur est de prendre pour valeur la plus grande des valeurs de ses voisins *plus* un. Sur l'exemple 4.2.(a), si $P_1$ agit, il prend la valeur 5. Si $P_4$ agit, il prend la valeur 3. Que le démon fasse agir $P_1$ puis $P_4$ ou $P_4$ puis $P_1$ n'a pas d'influence sur $C_2$. Par contre, sur l'exemple 4.2.(b), si $P_1$ agit avant $P_2$, $P_2$ prend pour valeur 6. Dans le cas contraire, $P_2$ prend pour valeur 3. Les deux processeurs $P_1$ et $P_2$ ne sont donc pas indépendants.

**Lemme 4.3.2**
Si $P_1$ et $P_2$ ne sont pas voisins l'un de l'autre, alors ils sont indépendants.

**Preuve :** Soit $C_0$ une configuration dans laquelle $P_1$ et $P_2$ sont activables. A priori, on a :

$$\begin{array}{ccc} & C_1 & C_2 \\ C_0 & & \\ & C'_1 & C'_2 \end{array}$$

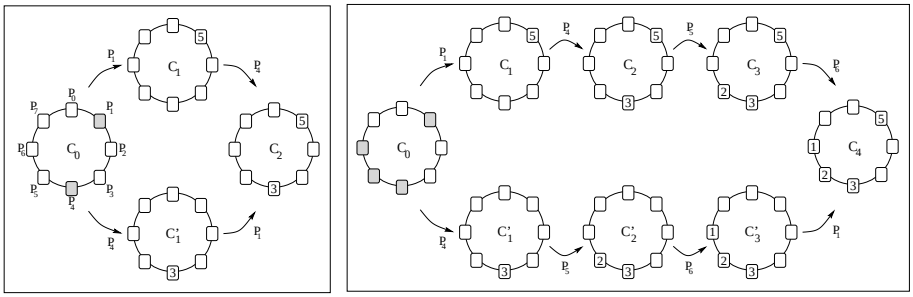



FIG. 4.3 – *Indépendance de processeurs*

Montrons que $C_2$ et $C'_2$ sont identiques.

$P_1$ n'étant pas un voisin de $P_2$, les voisins de $P_2$ ont même valeur dans $C_0$ et dans $C_1$. D'ou $P_2$ exécute la même action à partir de $C_0$ que de $C_1$ et $Variable_{C'_1}(P_2) = Variable_{C_2}(P_2)$. Comme la transition $C'_1 \xrightarrow{P_1} C'_2$ ne modifie pas la valeur de $P_2$, on a

$$Variable_{C'_2}(P_2) = Variable_{C_2}(P_2)$$

Le même raisonnement pour les voisins de $P_1$ donne $Variable_{C_2}(P_1) = Variable_{C'_2}(P_1)$. Comme $P_1$ et $P_2$ sont les seuls processeurs à avoir modifié leur valeur, on a $C_2 = C'_2$.

□

**Définition 4.3.3**
Soit $P$ un processeur et $\{P_i\}$ un groupe de processeurs. $P$ et $\{P_i\}$ sont indépendants si $P$ est indépendant de chacun des $P_i$.

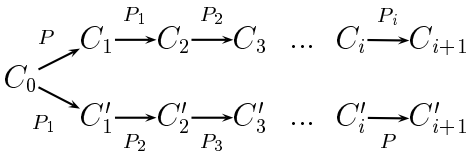
$$P \parallel \{P_i\} \Longleftrightarrow \forall i, P \parallel P_i$$

La figure 4.3 présente un exemple où un processeur ($P_1$) est indépendant d'un groupe de processeurs ($\{P_4$,$P_5$,$P_7\}$)

**Construction d'exécution** Nous allons maintenant construire progressivement une exécution ayant la "bonne" propriété de ne pas converger rapidement. Pour cela, nous montrons qu'il existe toujours une exécution dont les $i$ premiers processeurs à agir sont liés.

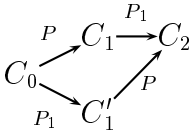
**Lemme 4.3.4**
Soit $\mathcal{E}$ une exécution du système. Soit $C_0 \xrightarrow{P} C_1$ la première transition de $\mathcal{E}$ et $C_1 \xrightarrow{P_1} C_2 \xrightarrow{P_2} C_3 \xrightarrow{P_3} \ldots \xrightarrow{P_i} C_{i+1}$ les $i$ transitions suivantes. Si $P$ et l'ensemble $\{P_i\}$ sont indépendants, alors l'exécution obtenue en faisant d'abord agir $P_1$, $P_2$, ..., $P_i$ puis seulement $P$ conduit aussi à la configuration $C_{i+1}$.

**Preuve :** A priori, on a :

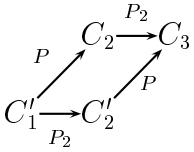


Montrons que $C_{i+1}$ et $C'_{i+1}$ sont identiques.

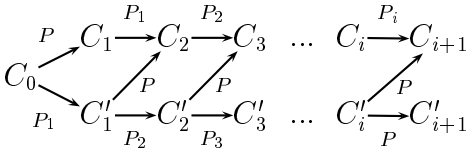
$P$ et $P_1$ sont indépendants. D'où l'ordre dans lequel ils agissent n'a pas d'influence sur l'exécution (lemme 4.3.2) et



Dans $C'_1$, $P$ et $P_2$ sont indépendants. A nouveau, l'ordre dans lequel ils agissent n'a pas d'importance et



De proche en proche, on obtient



Au final, on a bien $C_{i+1} = C'_{i+1}$ □

L'exemple 4.2(b) illustre ce lemme. $P_1$ est indépendant de $\{P_4,P_5,P_6\}$. Que $P_1$ agisse avant $P_4$, $P_5$ et $P_6$ ou que $P_4$, $P_5$ et $P_6$ agissent avant $P_1$ n'influe pas sur $C_4$.

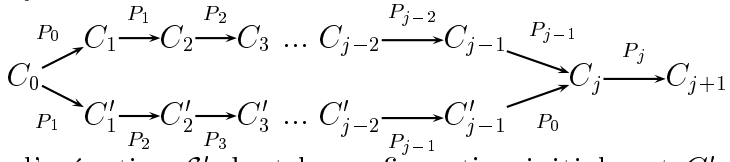
**Définition 4.3.5 : processeurs liés**
Soit $P$ et $P'$ deux processeurs. S'ils ne sont pas indépendants, on dit qu'ils sont liés et on note $P \not\parallel P'$. De même, un processeur et un ensemble de processeurs non indépendants sont liés.
Une suite de processeurs ($P_1$,$P_2$,$P_3$,...) est liée si chaque processeur $P_i$ de la suite est lié avec l'ensemble des processeurs le précédant.

$\forall i, P_i \not\parallel \{P_j\}_{j\in[1..i-1]}$

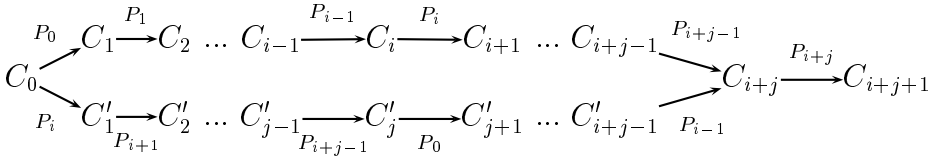
**Lemme 4.3.6**
Soit $\mathcal{S}$ un système réparti auto-stabilisant pour le problème de l'exclusion mutuelle. Alors il existe une exécution dont les $n-3$ premiers processeurs à agir sont liés.

**Preuve :** Montrons par récurence qu'il existe une exécution dont les $i$ premiers processeurs sont liés. Soit $\mathcal{E}$ une exécution du système.

1. Construisons une exécution dont les deux premiers processeurs sont liés : Soit $C_0 \xrightarrow{P_0} C_1$ la première transition de $\mathcal{E}$. Soit $P_j$ le premier processeur de $\mathcal{E}$ à être lié avec $P_0$ (éventuellement $P_0$ lui-même). Tous les processeurs situés entre $P_0$ et $P_j$ ne sont pas liés avec $P_0$. On a donc :



   Considérons l'exécution $\mathcal{E}'$ dont la configuration initiale est $C'_{j-1}$. Ses deux premières transitions sont $C'_{j-1} \xrightarrow{P_0} C_j \xrightarrow{P_j} C_{j+1}$. Les deux premiers processeurs a agir dans $\mathcal{E}'$ sont donc liés.
2. Supposons qu'il existe une exécution $\mathcal{E}$ dont les $i$ premiers processeurs soient liés.
3. Construisons $\mathcal{E}'$, une exécution dont les $i+1$ premiers processeurs sont liés. Soit $P_{i+j}$ le premier processeur lié avec $(P_0,P_1,...P_{i-1})$ ($P_{i+j}$ peut éventuellement être dans $\{P_0,P_1,...P_{i-1}\}$. Aucun des processeurs compris entre $P_{i-1}$ et $P_{i+j}$ n'est lié avec $(P_0,P_1,...P_{i-1})$. On a donc :



Considérons l'exécution $\mathcal{E}'$ dont la configuration initiale est $C'_j$. Ses $i+1$ premières transitions sont $C'_j \xrightarrow{P_0} C'_{j+1} \xrightarrow{P_1} C'_{j+2}...C'_{i+j-1} \xrightarrow{P_{i-1}} C'_{i+j} \xrightarrow{P_{i+j}} C'_{i+j+1}$. Ses $i+1$ premiers processeurs sont donc liés. □

**Lemme 4.3.7**
Soit $\mathcal{E}$ une exécution. Soit $(P_0,P_1,...,P_i)$ une suite de processeurs liés. Alors l'ensemble de processeurs $\{P_j\}_{j\in[0..i]}$ est connexe.

**Preuve :** Ce lemme se montre par récurence sur le nombre de processeurs de la suite.

1. Pour une suite de deux processeurs : les deux processeurs sont liés, ils sont donc voisins et forment un ensemble connexe (contraposé du lemme 4.3.2.
2. Supposons la propriété vraie pour une suite de $i$ processeurs ($i \geq 2$).
3. Montrons qu'elle est vraie pour $i+1$ processeurs. Soit $P_i$ le dernier processeur de la suite. Les $i-1$ premiers processeurs $\{P_j\}_{j\in[1..i-1]}$ forment une composante connexe. $P_i$ étant lié avec $\{P_j\}_{j\in[1..i-1]}$, il est lié avec au moins un des $P_j$. Ils sont donc voisins et l'ensemble $\{P_j\}_{j\in[1..i-1]} \cup P_i$ est connexe.

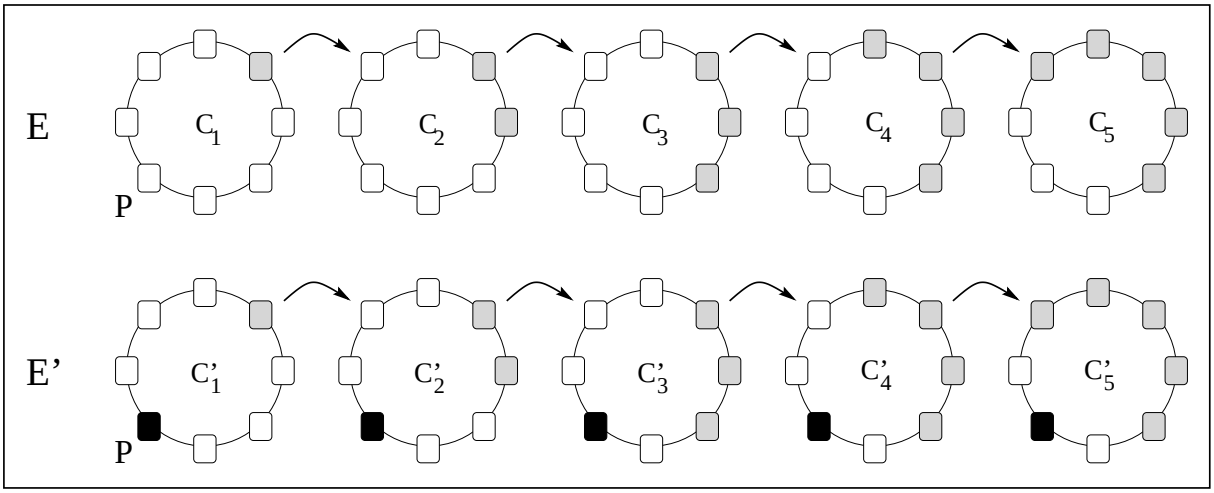



FIG. 4.4 – *Construction d'une exécution ayant une phase de stabilisation de n*

□

**Lemme 4.3.8**
Soit $C$ une configuration et $\mathcal{P}$ un ensemble connexe de $n-3$ processeurs. Alors il existe un processeur $P$ dont aucun des voisins n'est dans $\mathcal{P}$.

**Preuve :** Sur un anneau, un ensemble connexe est une chaîne. Le complémentaire d'un ensemble connexe est donc également connexe.

$\mathcal{P}$ étant connexe, il existe un ensemble connexe de 3 éléments. Cet ensemble est une chaîne et le processeur qui n'est pas une des extrémités de la chaîne vérifie le lemme. □

**Lemme 4.3.9**
Soit $\mathcal{S}$ un système auto-stabilisant pour l'exclusion mutuelle. Alors il existe une exécution de $\mathcal{S}$ ayant une phase de stabilisation d'au moins $n-2$ transitions.

**Preuve :** Soit $\mathcal{E}$ une exécution dont les $n-3$ premiers processeurs $(P_0,P_1,...,P_{n-4})$ forment une suite liée ($\mathcal{E}$ existe, lemme 4.3.6). Soit $P$ un processeur indépendant de $(P_0,P_1,...,P_{n-4})$ ($P$ existe, lemme 4.3.8). Soit $C_0$ la configuration initiale de $\mathcal{E}$. Soit $C'_0$ la configuration illégitime obtenue à partir de $C_0$ en corrompant le processeur $P$. Montrons qu'il existe une exécution ayant $C'_0$ pour configuration initiale et ne convergeant pas en moins de $n-3$ transitions.

Dans $C'_0$ le démon peut choisir de faire agir $P_0$ (car $P_0$ et ses voisins ont même valeur dans $C_0$ et dans $C'_0$). La transition $C'_0 \xrightarrow{P_0} C'_1$ conduit le système dans une configuration illegitime (car ni $P$, ni ses voisins n'ont changé de valeur). Dans $C'_1$, le démon peut choisir de faire agir $P_1$. Le système arrive alors dans une configuration illégitime $C'_2$. De proche en proche, après $n-3$ transitions, tous les processeurs de la suite $(P_0,P_1,...,P_{n-4})$ ont agi. Mais les valeurs de $P$ et de ses voisins n'ayant pas été modifiées, le système est toujours dans une configuration illégitime. □

Une illustration de ce lemme est donnée figure 4.4. Sur chaque configuration sont représentés en gris les processeurs qui agissent. A partir d'une exécution $\mathcal{E}$ dont les $n-3$ premiers processeurs à agir sont liés, on peut construire $\mathcal{E}'$ une exécution dans laquelle le démon ne permet pas au processeur corrompu ou à un de ses voisins d'agir avant $n-3$ transitions.

# Chapitre 5

# Procedure TEST

Dans le cas particulier de l'algorihtme de Dijkstra, la convergence en cas d'une faute unique n'est pas de l'ordre de $n$, mais de $3n$. Cela est dû au fait qu'une unique corruption fait apparaître deux privilèges sur l'anneau. Naturellement, le privilège qui existait avant la corruption peut ne pas avoir disparu. Au total, c'est donc trois privilèges que le démon peut faire avancer à tour de rôle. Et s'il fait en sorte qu'ils se rattrapent le plus tard possible, ils auront le temps de faire chacun un tour d'anneau avant la stabilisation.

Parallèlement à cela, rien ne permet à un processeur non corrompu de savoir si le privilège qu'il posséde est dû à une corruption ou non. Dans le doute, il le transmet, propageant potentiellement une faute locale au reste de l'anneau.

Dans ce chapitre, nous présentons deux techniques permettant d'améliorer cet état des choses. La première est l'utilisation d'un test qui du point de vue macroscopique peut-être présenté comme un oracle sélectionnant les privilèges à faire avancer et ceux à bloquer pour assurer une phase de stabilisation courte. La seconde consiste à forcer les processeurs à faire des demandes de confirmation sur les informations qu'ils viennent de recevoir.

## 5.1 Techniques d'amélioration

### 5.1.1 Test sur la légitimité d'un privilège

Au niveau plus élémentaire d'une simple transition, on se retrouve confronté au problème suivant : après une corruption de faible envergure, un certain nombre de privilèges sont présents sur l'anneau. Certains d'entre eux doivent leur existence à la corruption du processeur précédant leur support actuel ; ils ont des valeurs corrompues. Lorsqu'ils avancent, ils propagent leur fausse valeur, augmentant ainsi la distance entre la configuration actuelle d'une configuration légitime. D'autres au contraire existent parce qu'ils sont sur un processeur corrompu dont le prédécesseur n'a pas été corrompu. Ils ont eux une valeur correcte. Lorsqu'ils avancent, ils la propagent, corrigeant ainsi les variables de processeurs corrompus.

La configuration $C_0$ de l'exemple donnée figure 5.1 illustre les différents types de privilège. $C_0$ est obtenue en corrompant un processeur de la configuration légitime $L$.

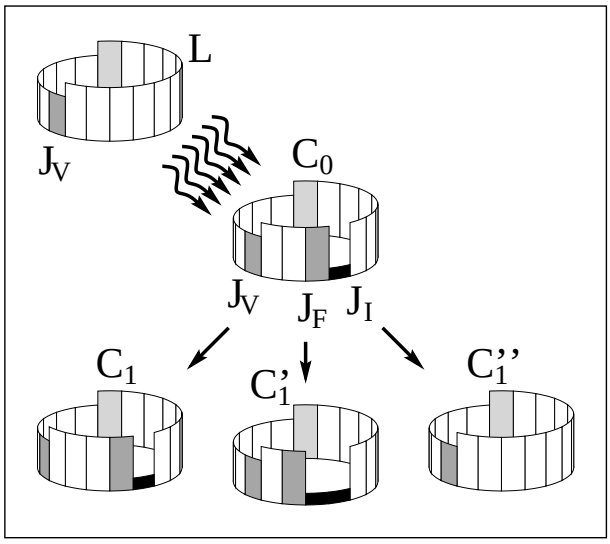



FIG. 5.1 – *Divers choix du démon*

Sa distance à $L$ est 1. $J_F$ est un privilège dans $C_0$ dont le prédécesseur est corrompu. $J_I$ est un privilège dont le prédécesseur n'est pas corrompu. Sa valeur est correcte.

A partir de $C_0$, trois choses sont possibles :

1. $C_0 \xrightarrow{J_V} C_1$ : si le processeur $J_V$ (appelé "vrai privilège" car c'est celui qui était présent dans la configuration légitime $L$) avance, le nombre de privilèges ne change pas, pas plus que la distance entre $C_1$ et l'ensemble des configurations légitimes.
2. $C_0 \xrightarrow{J_F} C'_1$ : $J_F$ doit son existence à la corruption du processeur précédant son support actuel. Ce genre de privilège est appele "faux privilège". Sa valeur est corrompue. Lorsqu'il avance, il propage sa fausse valeur (coloriée en noir sur la figure). Il n'y a toujours que trois privilèges sur l'anneau, mais la distance entre $C'_1$ et l'ensemble des configurations légitimes est maintenant de 2.
3. $C_0 \xrightarrow{J_I} C''_1$ : $J_F$ doit son existence à la corruption de son support. Ce genre de privilège est appelé "privilège correcteur". Son prédécesseur n'est pas corrompu. Lorsqu'il avance, il rattrape $J_F$. La corruption est corrigée et les deux privilèges disparaissent. $C''_1$ est une configuration légitime.

Intuitivement, un algorithme où les privilèges faux ne pourraient pas avancer aurait un temps de convergence faible. La seule possibilité du démon serait alors de faire avancer des privilèges correcteurs, ce qui corrige des corruptions, ou de faire avancer le privilège vrai, ce qui n'aggrave pas les choses. Mais pour obtenir un tel résultat, encore faut-il que les privilèges puissent déterminer leur nature. Cela est rendu possible grâce à une propriété des privilèges et des corruptions :

**Propriétés 5.1.1**
Si un anneau compte $k$ processeurs corrompus, un privilège faux a toujours un privilège vrai ou correcteur parmi ses $k$ prédécesseurs.

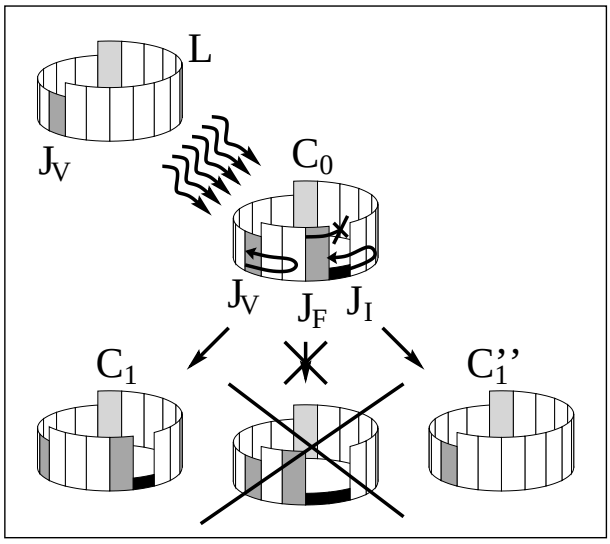



FIG. 5.2 – *Divers choix du démon restreint par un oracle*

Cela est dû au fait qu'une zone de processeurs corrompus est toujours encadrée par un faux privilège (en tête) et un privilège vrai ou correcteur (en queue).

Grâce à cette propriété, il va être possible de subordonner les déplacements de privilège au succès d'une procédure déterminant leur nature.

Pour cela, on demande aux processeurs privilégiés d'exécuter une procédure de détection avant de transmettre leur privilège. En première approximation, $TEST(P,k,NonPrivilege)$ peut-être considéré comme un oracle délivrant au processeur $P$ l'autorisation de transmettre son privilège seulement si aucun privilège n'est detecté parmi ses $k$ prédécesseurs (c'est à dire si les $k$ prédécesseurs de $P$ ont tous leur prédicat $NonPrivilege$ à vrai). Dans le cas contraire, l'oracle ne donne pas de réponse. Appliqué aux trois différents type de privilèges, on obtient :

1. Si $J$, le privilège de $P$, est un vrai privilège ou un privilège correcteur : lors de son exécution, $TEST(P,k,NonPrivilege)$ ne détecte aucun privilège parmi les $k$ prédécesseurs $P$. Il donne donc à $P$ l'autorisation de transmettre $J$.
2. Si $J$ est un faux privilège : nécessairement, un de ses $k$ prédécesseurs possède un privilège. Lors de son exécution, $TEST(P,k,NonPrivilege)$ le détecte et ne délivre pas d'autorisation de mouvement.

$TEST(P,k,NonPrivilege)$ empêche donc les faux privilèges de se déplacer, permettant ainsi aux privilèges vrais et aux correcteurs de corriger les processeurs corrompus.

Un exemple d'utilisation du test est donné figure 5.2. $J_V$ initie un test qui revient et lui apporte une autoristation de mouvement. $J_C$ fait de même. Par contre, la procédure de test appliquée à $J_F$ détecte un privilège parmi les prédécesseurs proches de $J_F$. La procédure n'aboutit pas et $J_F$ ne reçoit pas d'autorisation de mouvement.

Cette première technique permet d'améliorer le temps de convergence quand le nombre de fautes frappant l'anneau est faible puisqu'il passe de $O(n^2)$ (algorithme de Dijkstra) à $O(kn)$ (avec $k$, nombre de fautes frappant le réseau).

### 5.1.2 Anti-corruption

Nouveau retour à l'algorihtme de Dijkstra. Lorsqu'un processeur est corrompu, rien ne lui permet de le savoir. De même, son successeur n'a pas non plus l'information. Une technique telle que celle présentée ci-dessus permet d'éviter une propagation des erreurs, mais le mécanisme n'est pas parfait : en effet, les corruptions peuvent affecter toutes les variables d'un processeur. La réponse à la procédure de détection $TEST$ étant stockée, elle peut-être elle-même corrompue. Dans ce cas, un processeur ne peut plus faire la différence entre un privilège vrai et un privilège faux ayant une autorisation de transmettre.

D'une manière plus générale, même si l'algorithme demande aux processeurs de faire un grand nombre de vérifications avant d'autoriser une transmission de valeur, il est toujours possible qu'une corruption donne à un processeur exactement l'état précédant la transmission des données. Tout mécanisme de contrôle unilatéral est donc voué à l'échec.

Par contre, un contrôle bilatéral permet de faire barrage aux corruptions. L'idée semble assez naturelle : avant de recevoir une autorisation de transmettre un privilège, un processeur doit en faire la demande. Cela permet d'éviter qu'une réponse soit donnée à une question ne lui correspondant pas. Aussi, avant d'initier un test, le processeur privilégié doit attendre que ses voisins soient disponibles. Ensuite, et ensuite seulement, il leur demande des vérifications. En prenant le même genre de précaution, les voisins s'assurent eux aussi qu'ils obtiennent des informations fiables. Au final, si le possesseur du privilège reçoit une autorisation, il sait qu'elle répond à sa question et il transmet le privilège. Dans le cas contraire, s'il ne reçoit pas de réponse ou s'il en reçoit une sans être à l'origine d'un test, il ne fait rien.

Pour résumer, une transmission de privilège doit se dérouler en quatre étapes :

1. Un processeur souhaitant transmettre son privilège doit attendre que son prédécesseur soit prêt à exécuter le test (pas de test en cours ou terminé).
2. Quand son prédécesseur est prêt, il fait une demande d'autorisation.
3. Le prédécesseur exécute alors le test.
4. Si le test apporte une réponse positive, le processeur transmet son privilège.

Dernier écueil, une corruption peut intervenir après l'étape trois : juste après que $P$ ait fait une demande d'autorisation, une corruption frappe $P^-$, modifiant sa valeur et lui faisant croire à une réponse positive. Il est à noter que ce problème ne peut affecter que le privilége vrai. En effet, dans une configuration légitime, il est le seul privilège existant. Il est donc le seul à pouvoir être en train de faire une demande d'autorisation au moment où une faute frappe. Pour éviter cela, le processeur désirant transmettre un privilège doit adapter le comportement suivant : en même temps qu'il demande à son prédécesseur de faire le test, il note la valeur qu'il est sensé prendre en cas de réponse positive. Lorsque son prédécesseur lui transmet une réponse, il vérifie que la valeur de son privilège n'a pas changé. Si c'est bien le cas, il le transmet. Sinon, il annule sa demande et recommence la procédure de transmission en bonne et due forme.

Dernier point, ce genre de procédure empêche les corruptions de se propager, mais rend l'algorithme extrêmenent vulnérable aux blocages. Aussi, à moins de disposer d'un

mécanisme de détection des blocages, le nombre de fautes toléré par l'algorithme ne peut pas être trop important. Dans notre cas, $\sqrt{n}-1$ fautes peuvent provoquer un blocage général. Aussi, l'algorithme est $k$-stabilisant pour $k < \sqrt{n}-1$

Un résumé des différentes techniques présenté ici est illustré figure 5.3. Dans l'algorithme de Dijkstra, un processeur agit sans effectuer de contrôle. La valeur qu'il prend peut être ou ne pas être une corruption, le processeur agit de même (figure 5.3.(a)). L'utilisation d'un test permet au processeur de prendre ou de ne pas prendre la valeur de son prédécesseur (figure 5.3.(b)). Cette technique est assez fiable. Elle permet même d'éviter certaines corruptions (le petit éclair symbolise une corruption qui vient de se produire) attenant aux réponses de tests (figure 5.3.(c)). Mais elle ne permet pas de protéger le vrai privilégié d'une corruption (figure 5.3.(d)). La technique anti-corruption y remedie (figure 5.3.(e)).

transmettre $J$, il interroge l'oracle. Si aucun de ses $k$ prédécesseurs n'a de privilège, l'oracle donne à $P$ une autorisation de transmission. Sinon, l'oracle ne répond pas.

De manière plus formelle, le test $TEST(P,k,NonPrivilege)$ fonctionne de la manière suivante :

**1.** Un processeur $P$ désireux de transmettre son privilège doit d'abord en demander l'autorisation, c'est à dire s'assurer qu'aucun de ses $k$ prédécesseurs ne possède également de privilège. On dit qu'il fait un *test à distance k*. Les communications étant limitées au voisinage immédiat, il ne peut le faire seul. Il va donc charger son prédécesseur $P^-$ de faire la vérification à sa place. Plus précisement, il attend que $P^-$ soit disponible (non impliqué dans un test), il stocke dans une variable la valeur qu'il pense devoir prendre et signale alors à $P^-$ qu'il entame un *test à distance k*.

**2.** Si $P^-$ a un privilège, il ne donne pas suite à la demande et le test initié par $P$ échoue. Sinon, si $NonPrivilege(P^-)$ est vrai, il a pour rôle de vérifier qu'aucun privilège n'est présent parmi ses $k-1$ prédécesseurs. Pour cela, il attend que $P^{--}$ soit disponible puis lui signale qu'il commence un *test à distance* $k-1$.

**3.** A son tour, si $P^{--}$ a un privilège, il ne répond pas et le test est bloqué. S'il n'en a pas, il attend que $P^{---}$ soit disponible puis entame un *test à distance* $k-2$.

**...** Ainsi de suite. La demande de test se transmet de la même manière jusqu'au $k-1^{ieme}$ prédécesseur de $P$.

**k.** Quand $P^{(k-1)-}$ reçoit la demande d'autorisation de $P^{(k-2)-}$, il attend que $P^{(k)-}$ soit disponible puis commence un *test à distance* 1.

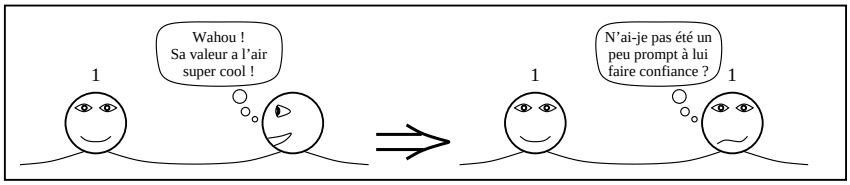
**k+1.** $P^{(k)-}$ est le dernier processeur à être impliqué dans le test. S'il a un privilège, il ne donne pas de suite au test. Sinon, il répond positivement à la demande de $P^{(k-1)-}$ et lui donne son autorisation.

**k+2.** Quand $P^{(k-1)-}$ reçoit la réponse positive de $P^{(k)-}$, il la transmet à $P^{(k-2)-}$.

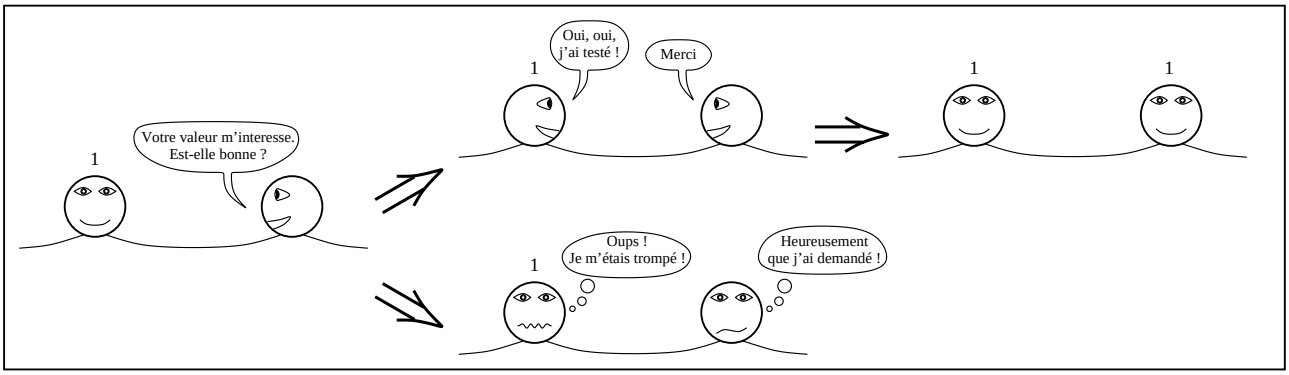
**...** Ainsi de suite jusqu'à $P^-$.

**2k.** Au final, quand $P^-$ reçoit la réponse positive de $P^{--}$, il donne à $P$ son "autorisation de mouvement".

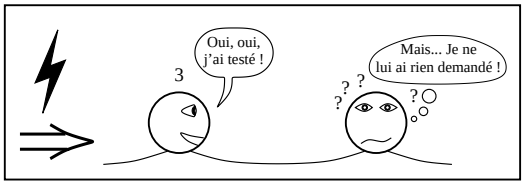
**2k+1.** $P$ peut alors transmettre son privilège.



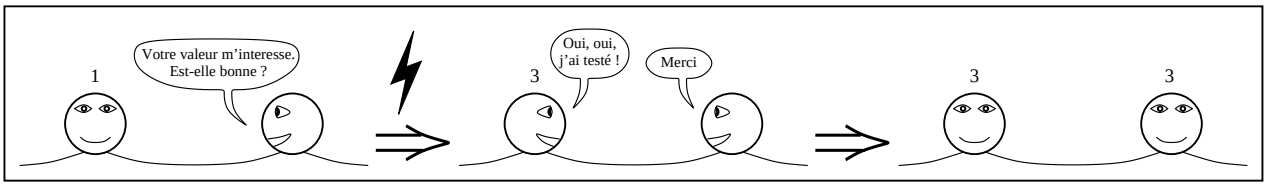

(a) Algorithme de Dijkstra



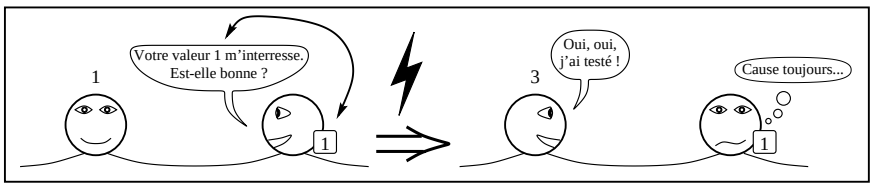

(b) Utilisation du Test anti-propagation




(c) Le Test doit être initié par le demandeur




(d) Faiblesse du Test




(e) Technique anti-corruption

FIG. 5.3 – *La vie d'un processeur au quotidien*

### 5.1.3 Implémentation

Comme nous venons de le voir, le test se déroule en plusieurs phases. La variable $Test(P)$ a pour rôle d'indiquer au processeur $P$ et à ses voisins la phase de test dans laquelle $P$ est engagé. Pour simplifier les notations, $Test(P)$ désignera le contenu de la variable $Test$ de $P$, $Test(P,P')$ sera le couple $(Test(P),Test(P'))$ et $Test(P,P',P'')$ sera le triplet $(Test(P),Test(P'),Test(P''))$. Lorsqu'il sera question plus spécifiquement d'un processeur $P_0$, sa valeur sera noté en gras. Par exemple, $Test(P_0^-,P_0,P_0^+)$ sera noté $(Test(P_0^-),\mathbf{Test(P_0)},Test(P_0^+))$.

A la base, un processeur non engagé dans le test attend. Dans ce cas, sa variable $Test(P)$ est à $A$, indiquant ainsi à ses voisins qu'il est disponible pour participer à un test. C'est également la valeur prise par un processeur détectant un certain type d'incohérence parmi les variables $Test$ de son voisinage (par exemple, s'il est en train de répondre a son successeur alors que celui-ci ne lui a pas posé de question).

Lorsqu'il veut transmettre son privilège, $P$ vérifie que $P^-$ est en *attente* puis lui demande une autorisation de transmission. Pour éviter les corruptions de réponse de test (section 5.1.2, page 80), il mémorise également la valeur de son privilège. Il fait tout cela en positionnant sa variable $Test$ à $Q_k^j$, avec $j = Priv(P^-)$, valeur du privilège de $P$. Mettre $Test$ à $Q$ signale que $P$ pose une question ("Puis-je recevoir une autorisation de mouvement ?"), $k$ indique le nombre de prédécesseurs à tester avant que la réponse revienne et $j$ permet de démasquer les fausses autorisations.

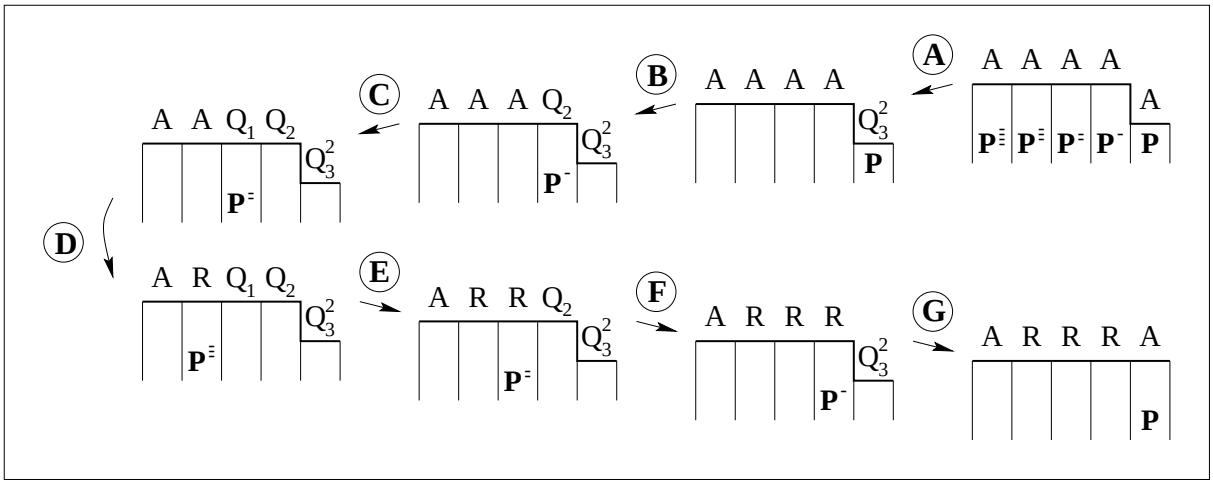
Lorsqu'un processeur $P$ non privilégié reçoit une demande d'autorisation, il doit la transmettre à son prédécesseur, après avoir vérifié que $P^-$ est en attente. Pour cela, il positionne sa variable $Test$ à $Q_i$. $Q$ signale que $P$ pose une question et $i$ indique le nombre de prédécesseurs à tester.

Lorsqu'un processeur $P$ non privilégié reçoit une demande d'autorisation ne concernant qu'un seul processeur (lui-même), il doit, s'il n'a pas de privilège, y répondre. Pour cela, il positionne sa variable $Test$ à $R$. $R$ constitue une réponse positive a une question posée.

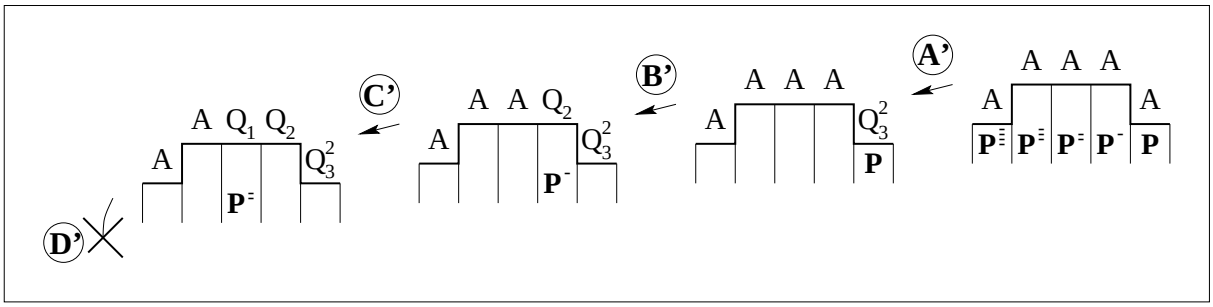
Lorsqu'un processeur a transmis une demande d'autorisation et qu'une réponse lui parvient, il doit la transmettre. Pour cela, il positionne sa variable $Test(P)$ à $R$. Lorsque le processeur en question est $P^-$, le test est achevé, $P$ a son autorisation de transmettre.

Un exemple de fonctionnement du test est donné figure 5.4. On suppose qu'au plus trois processeurs peuvent être corrompus ($k = 3$) et que la valeur du privilège de $P$ est 2.

1. **Transition A** et **A' :** le processeur $P$ est privilégié. $P^-$ étant en attente, $P$ initie donc un test chargé de vérifier qu'il n'y a pas de privilège parmi ses trois prédécesseurs. Pour cela, il positionne sa variable $Test(P)$ à $Q_3^2$ (2 étant la valeur de $Priv(P^-)$).
2. **Transition B** et **B' :** $P^-$ transmet la question de $P$ en posant $Test(P^-) = Q_2$.
3. **Transition C** et **C' :** à son tour $P^{--}$ transmet la question de $P^-$ en posant $Test(P^{--}) = Q_1$.
4. **Transition D :** $P^{---}$ est le dernier processeur à participer au test. Dans l'exemple 5.4(a), il n'a pas de privilège. Il répond donc positivement à $P^{--}$ en positionnant $Test(P^{--})$ à $R$.




(a) P a un token vrai ou illusoire




(b) P a un faux token

FIG. 5.4 – *Exemple d'exécution du test.*

**Prédicat : Définition d'un Privilège**

$$Privilege(P) \Leftrightarrow \begin{cases} Priv(P) \neq Priv(P^-) + 1 & \text{si } P \text{ est le processeur distingué.} \\ Priv(P) \neq Priv(P^-) & \text{si } P \text{ est un processeur quelconque.} \end{cases}$$

**Prédicats pour un privilégié**

$Question(P,k) \quad \Leftrightarrow Test(P^-,P) = (A,A)$

$TestEnCours(P) \quad \Leftrightarrow Test(P^-,P) = (A,Q_k^j)$ ou $Test(P^-,P) = (Q_{k-1},Q_k^j)$ avec $j = Priv(P^-)$

$Autorisation(P) \quad \Leftrightarrow Test(P^-,P) = (R,Q_k^j)$ avec $j = Priv(P^-)$

$Probleme(P) \quad \Leftrightarrow \neg Question(P,k) \wedge \neg TestEnCours(P) \wedge \neg Autorisation(P)$

**Prédicats pour un non privilégié**

$Question(P,i) \quad \Leftrightarrow Test(P^-,P,P^+) = (A,A,Q_{i+1})$

$Reponse(P,1) \quad \Leftrightarrow Test(P^-,P,P^+) = (A,A,Q_1)$ ou $Test(P^-,P,P^+) = (Q_k^j,A,Q_1)$

$Reponse(P,i+1) \quad \Leftrightarrow Test(P^-,P,P^+) = (R,Q_i,Q_{i+1})$

$Probleme(P) \quad \Leftrightarrow Test(P,P^+)) \in \{(R,A),(Q_i,A),(Q_i,R),(Q_i,Q_{i'})\}$

**Règles gardées**

$R1$ $Privilege(P)$ et $Autorisation(P)$ $\Leftrightarrow Priv(P) \longleftarrow \begin{cases} Priv(P^-)+1 & \text{Si } P \text{ est le distingué.} \\ Priv(P^-) & \text{Sinon.} \end{cases}$

$R2$ $Question(P,i)$ $\Leftrightarrow Test(P) \longleftarrow Q_i$

$R3$ $Reponse(P,i)$ $\Leftrightarrow Test(P) \longleftarrow R$

$R4$ $Probleme(P)$ $\Leftrightarrow Test(P) \longleftarrow A$

FIG. 5.5 – *Exclusion mutuelle k-stabilisante*

5. **Transition D' :** dans l'exemple 5.4(b), $P^{---}$ a un privilège. Il ne donne donc pas d'autorisation. La demande d'autorisation de $P$ n'aboutit pas et le test échoue.
6. **Transition E :** $P^{--}$ a reçu une réponse positive, il la transmet à $P^-$.
7. **Transition F :** $P^-$ a reçu une réponse positive, il donne donc à $P$ une autorisation de mouvement.
8. **Transition G :** $P$ fort de son autorisation, $P$ transmet son privilège à son successeur.

### 5.1.4 Le protocole

Les prédicats définissent les conditions à réunir pour qu'un processeur puisse agir.

1. **Privilege(P)** est vrai quand $P$ a un privilège. Le distingué $D$ a le privilège quand sa variable $Priv(D)$ est différente de celle de son prédécesseur $D^-$ *plus un*, les autres processeurs l'ont quand leur variable $Priv(P)$ est différente de celle de leur prédécesseur.
2. Pour un processeur possédant un privilège, on définit les prédicats suivants :
   (a) **Question(P,k)** est vrai quand $P$ est prêt à faire une demande d'autorisation ($Test(P) = A$) et que $P^-$ est prêt à la transmettre ($Test(P^-) = A$).
   (b) **TestEnCours(P)** est vrai quand $P$ a fait une demande ($Test(P) = Q_k^j$) et que $P^-$ va la transmettre ($Test(P^-) = A$), ou l'a transmise ($Test(P^-) = Q_{k-1}$).

(c) **Autorisation(P)** est vrai quand $P$ a fait une demande d'autorisation ($Test(P) = Q_k^j$), que $P^-$ y a répondu positivement ($Test(P^-) = R$) et que la valeur du privilège est la même que lors de la demande d'autorisation ($j = Priv(P^-)$).

(d) **Probleme** est vrai si $P$ détecte un problème dans la configuration des variables $Test(P)$ et $Test(P^-)$, par exemple, si une réponse arrive sans qu'une question soit posée, ou si une question posée par $P$ ne correspond pas à la question transmise par $P^-$.

3. Pour les processeurs n'ayant pas de privilège, on définit trois prédicats :

(a) **Question(P,i)** est vrai quand $P$ est en attente ($Test(P) = A$), n'a pas de privilège, qu'il reçoit de $P^+$ une demande d'autorisation ($Test(P^+) = Q_{i+1}$) et que $P^-$ est prêt à transmettre la demande ($Test(P^-) = A$).

(b) **Reponse(P)** est vrai dans deux cas : quand $P$ reçoit une demande et qu'il est le dernier processeur à participer au test ($Test(P^+) = Q_1$), $Reponse(P)$ est vrai si $P^-$ est en attente ou si $P^-$ est un privilègié en train de faire un test ($Test(P^-) = Q_k^j$). $Reponse(P)$ est également vrai quand $P^-$ répond à la question ($Test(P^-) = R$) que $P$ lui avait précédemment posée ($Test(P,P^+) = Q_i,Q_{i+1}$).

(c) **Probleme** est vrai quand $P$ détecte certains types de problème. Plus précisement, une réponse sans un successeur posant une question ($Test(P,P^+) = (R,A)$) ou une question sans un successeur posant une question cohérente ($Test(P,P^+) \in \{(Q_i,A),(Q_i,R),(Q_i,Q_{i'})\}$) sont considérées ici comme des problèmes [1].

Ces prédicats permettent de définir les règles gardées :

1. $R1$ : quand le privilègié en a l'autorisation, il transmet son privilège. Comme pour l'algorithme de Dijkstra, il fait cela en modifiant la valeur de sa variable $Priv(P)$.
2. $R2$ : quand tout est prêt pour qu'une demande d'autorisation soit transmise, $P$ la transmet.
3. $R3$ : quand tout est prêt pour qu'une réponse soit transmise, $P$ la transmet.
4. $R4$ : quand $P$ detecte une situation anormale, il se met en attente.

## 5.2 Preuve

La preuve de l'algorithme nécessite l'introduction de quelques définitions.

1. Il existe bien d'autres situations qui pourraient être considérées comme irrégulières. Mais, pour un processeur donné, il est plus simple et plus efficace de ne corriger que les situations irrégulières dûes à son successeur. Par exemple, si $Test(P^-,P,P^+) = (Q_2,Q_5,Q_6)$, le processeur $P$ sait que quelque chose ne va pas. Mais $P^-$ s'en rend compte également. D'où plutôt que de permettre à $P$ et $P^-$ de faire des corrections (et de laisser ainsi au démon un plus grand choix pour allonger la phase de stabilisation), l'algorithme tranche : seul $P^-$ peut faire les modifications qui s'imposent. Aussi, $Probleme(P)$ ne detecte pas tous les problèmes, mais seulement ceux qui doivent être corrigés par $P$.

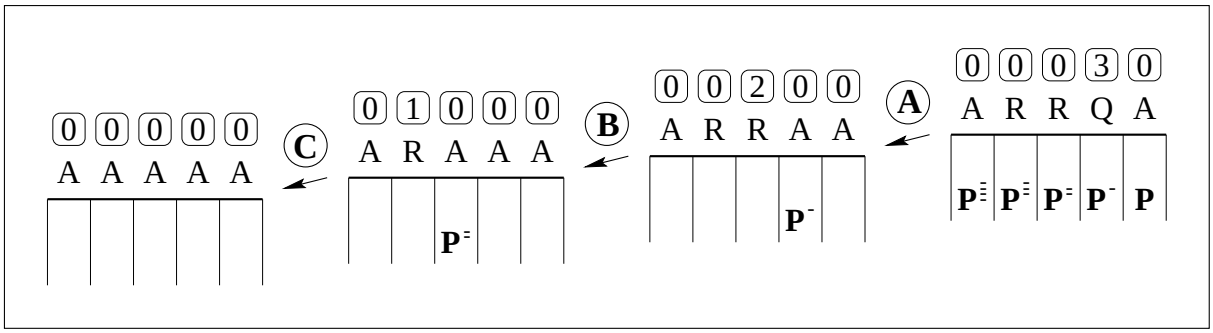



FIG. 5.6 – *Exemple de calcul de* $Chaine_{RQ}$

### 5.2.1 Définition

Dans une configuration illégitime, certains processeurs ont leur prédicat *Probleme* à vrai. Ces processeurs peuvent appliquer la règle $R4$. Ce faisant, ils initialisent leur variable $Test$, se préparant ainsi à une future exécution du test.

**Définition 5.2.1**
Soit $C$ une configuration. On dit qu'un processeur est un *initialisateur de test* s'il a son prédicat *Probleme* à vrai. L'ensemble de tous les initialisateurs de test est noté $Probleme(C)$.

Informellement, un initialisateur de test $P$ qui applique la règle $R4$ peut transformer son prédécesseur immédiat $P^-$ en initialisateur de test. Celui-ci pensait qu'il était dans une situation correcte, mais par l'action de $P$, il voit son prédicat *Probleme* devenir vrai. A son tour, il devient initialisateur de test et peut appliquer $R4$ (c'est à dire se mettre en attente). Ainsi, de proche en proche, la mise en attente des processeurs se propage. $Chaine(P)$ est le nombre de prédécesseurs d'un initialisateur de test qui sont susceptibles de faire partie de la "vague d'initialisation" déclenchée par l'action de $P$.

**Définition 5.2.2**
Soit $C$ une configuration et $P$ un processeur de $C$. $Chaine_{RQ}(P)$ est le nombre de prédécesseurs de $P$ n'étant ni des privilégiés, ni des initialisateurs de test et ayant leur variable $Test$ à $R$ ou à $Q_i$:

$$Ch_{RQ}(P) = \begin{cases} 0 & \text{si } Test(P) = A \\ 0 & \text{si } P \text{ est priviligié} \\ 0 & \text{si } P \text{ est dans } Probleme(C) \\ 1 + Ch_{RQ}(P^-) & \text{sinon} \end{cases}$$

Un exemple est présenté figure 5.6. Pour chaque processeur, la valeur de sa $chaine_{RQ}$ est donnée au-dessus, dans un cadre. Dans la première configuration (la plus à droite), $P^-$ est un initialisateur de test. Il met sa valeur à 0, $P^{--}$ devient à son tour un initialisateur de test. Dans la première configuration, $Ch_{RQ}(P^-)$ vaut 3 car dans la suite de l'exécution, 3 processeurs vont tour à tour devenir des initialisateurs.

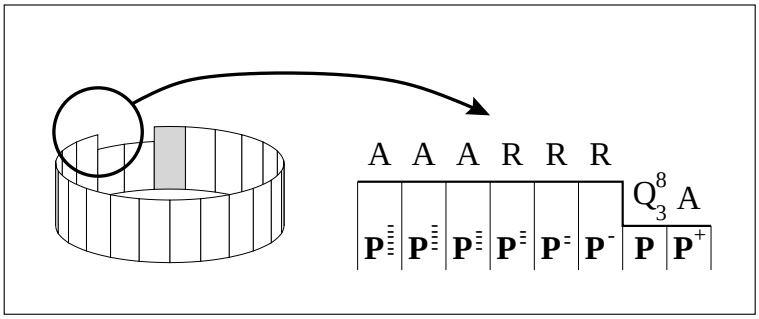



FIG. 5.7 – *Exemple de configuration légitime*

### 5.2.2 Configurations légitimes

**Définition 5.2.3 : Configuration légitime**
Une configuration $L$ est légitime si :

1. Il existe un unique processeur $P$ ayant un privilège dans $L$.
2. $P$ a fait une demande d'autorisation de mouvement ($Test(P) = Q_k^j$) avec $j = Priv(P^-)$).
3. Les $k$ prédécesseurs de $P$ ont répondu positivement à la demande de $P$ (pour tout $i \in [1..k]$, on a $Test(P^{(i)-}) = R$).
4. Tous les autres processeurs $P'$ sont au repos ($Test(P') = A$).

Un exemple de configuration légitime est donné figure 5.7 (avec $k = 3$).

### 5.2.3 Fonction potentiel du test

La preuve de la $k$-convergence comme celle de la correction utilisent toutes deux une fonction $F3$ appelée fonction de "convergence de test". $F3$ a pour particularité d'être positive et décroissante sur toutes transitions n'impliquant pas de mouvement de privilège, c'est à dire sur toutes les transitions participant à l'exécution d'un Test. Elle permet de montrer que, au cours d'une exécution, toute séquence de transitions n'impliquant pas de mouvement de privilège est finie.

Pratiquement, $F3$ est une fonction de $\mathcal{C}$ dans $\mathbb{N}$ associant un entier naturel à chaque configuration $C$.

D'un point de vue théorique, $F3$ peut être considéré comme un potentiel associé aux tests. En effet, considérons une portion d'exécution dans laquelle aucune transition ne fait bouger de privilège. Considérons également un test $T$, initié par un processeur $P$, en cours d'exécution. Il est possible de déterminer localement le nombre de transitions minimum avant qu'une réponse positive ne soit donnée à $P$. Par exemple, si $Test(P') = Q_i$ et $Test(P'^-) = A$, avant qu'une réponse ne parvienne à $P$, il faut qu'une demande d'autorisation soit transmise à $P'^-$, à $P'^{--}$, ..., à $P'^{i-}$ puis que la réponse revienne vers $P$. Cela fait un total de $i + k$ transitions. Tout cela a pu être determiné uniquement en considérant l'état des deux processeurs $P'$ et $P'^-$. $i + k$ peut donc être considéré comme

le *potentiel* du couple ($P^-$,$P$) : au cours de l'exécution à suivre, les valeurs actuelles des variables de ($P^-$,$P$) seront la cause de $k+i$ transitions.

Concrètement, $F4$ est une fonction qui à chaque couple ($P^-$,$P$) associe un potentiel. Etant donnée une configuration $C$, $F3$ est alors la somme des potentiels $F4(P^-,P)$ pour tous les couples de processeurs de $C$.

**Notations :** De manière générale, $i$ et $i'$ désignent deux entiers variant de 1 à $k$ tel que $i' \neq i+1$.

Pour un processeur privilégié $P$, $j$ est la valeur du privilège de $P$ alors que $j'$ est une valeur quelconque différente de $j$. A partir de là, $\tilde{Q}$ désigne la valeur $Q_k^j$ lorsque le prédicat $Probleme(P)$ est faux. A l'inverse, $Q'$ désigne soit la valeur $Q_i$ avec $i \neq k$, soit $Q_k^{j'}$, soit $Q_k^j$ lorsque $Probleme(P)$ est vrai.

Pour un processeur quelconque (y compris un processeur privilégié), $?_R^Q$ désigne soit la variable $R$, soit $Q_i$, soit $Q'$ (pour un privilégié).

**Définition 5.2.4**
Soit ($P^-$,$P$) un couple processeur. La fonction $F4$ est appelée *potentiel* du couple ($P^-$,$P$). Elle dépend de la présence éventuelle d'un privilège en $P$, en $P^-$ et de l'état des variables de $P^-$ et $P$ :

| **Si ni $P^-$, ni $P$ n'ont de privilège** | | |
|---|---|---|
| **(Test($P^-$,$P$))** | $\longmapsto$ | **F($P^-$,$P$)** |
| ($A$,$Q_i$) | $\longmapsto$ | $k+2(i)-1$ |
| ($R$,$Q_i$) | $\longmapsto$ | $k-(i)$ |
| ($A$,$A$) | $\longmapsto$ | 0 |
| (?,$A$) | $\longmapsto$ | $Ch_{RQ}(P^-)$ |
| ($Q_i$,$Q_{i'}$) | $\longmapsto$ | $3k+Ch_{RQ}(P)$ |
| Autres | $\longmapsto$ | 0 |

| **Si $P$ a un privilège** | | |
|---|---|---|
| **(Test($P^-$,$P$))** | $\longmapsto$ | **F($P^-$,$P$)** |
| ($A$,$Q_k^j$) | $\longmapsto$ | $3k-1$ |
| ($Q_{k-1}$,$Q_k^j$) | $\longmapsto$ | 0 |
| ($R$,$Q_k^j$) | $\longmapsto$ | 0 |
| ($A$,$Q_1$) | $\longmapsto$ | $3k+3$ |
| (?,$A$) | $\longmapsto$ | $3k+Ch_{RQ}(P^-)$ |
| Autres | $\longmapsto$ | $3k+Ch_{RQ}(P)$ |

| **Si $P^-$ a un privilège** | | |
|---|---|---|
| **(Test($P^-$,$P$))** | $\longmapsto$ | **F($P^-$,$P$)** |
| ($R$,$Q_i$) | $\longmapsto$ | $k-(i)$ |
| Autres | $\longmapsto$ | 0 |

| **Si $P^-$ et $P$ ont un privilège** | | |
|---|---|---|
| **(Test($P^-$,$P$))** | $\longmapsto$ | **F($P^-$,$P$)** |
| ($\tilde{Q}$,$A$) | $\longmapsto$ | 0 |
| ($\tilde{Q}$,$G$) | $\longmapsto$ | 1 |
| ($A$,$\tilde{Q}$) | $\longmapsto$ | 1 |
| ($A$,$A$) | $\longmapsto$ | 2 |
| ($R$,$A$) | $\longmapsto$ | 2 |
| Autres | $\longmapsto$ | 2 |

$F3$ est la somme des potentiels de tous les couples ($P^-$,$P$) pour une configuration donnée : $F3(C) = \sum_{(P^-,P)\in C} F4(P^-,P)$

**Note :** Certaines valeurs sont définies plusieurs fois. Par exemple, $F4$ du couple ($A$,$A$), lorsque ni $P^-$, ni $P$ ne sont privilégiés, est définie comme $F4(A,A) = 0$ ou comme $F4(?,A) = Ch_{RQ}(P^-)$. Dans ce cas, la valeur à utiliser est la première valeur rencontrée

dans le tableau (en le parcourant de haut en bas). (?,$A$) doit être considéré comme "tous les couples dont le deuxième élément est $A$ sauf ceux déjà définis précédemment".

La figure 5.8 donne deux exemples d'évaluation partielle des fonctions $F3$ et $F4$ au cours de l'exécution d'un test. Pour chaque couple de processeurs, la valeur de $F4$ est donnée en encadré au dessus du couple. $F3$ est la somme de tous les $F4$. Le premier exemple concerne une exécution légitime, le second une exécution quelconque. Dans les deux cas, $F3$ décroît sur chaque transition car aucune d'entre elles n'implique de transmission de privilège. Cette caractéristique est en fait la principale propriété de $F3$, propriété que nous allons maintenant démontrer.

**Remarque préliminaire 5.2.5**
La présentation de l'algorithme donnée figure 5.5 utilise de nombreux prédicats. Cette écriture facilite la lecture et la compréhention des règles, mais nous éloigne de la réalité de l'algorithme. Aussi, pour simplifier les preuves à suivre, il nous à semblé intéressant de redonner les règles de l'algorithme en les présentant sous leur forme brute, sans utiliser de prédicat (à part $Privilege$). Elles sont présentées figure 5.9.
Les règles sont données pour un processeur $P$. Nous rappelons que le triplet ($x$,$\mathbf{y}$,$z$) désigne ($Test(P^-)$,$\mathbf{Test(P)}$,$Test(P^+)$), avec la valeur de $P$ en gras au centre.

**Lemme 5.2.6**
$F3$ est strictement décroissante sur toute transition concernant l'exécution du test.

**Preuve :** Soit $T = C \xrightarrow{R} C'$ une transition avec $R \neq R1$. Soit $\Delta F3(T) = F3(C') - F3(C)$. Montrons que $\Delta F3(T)$ est strictement négative.

$F3$ est calculé en sommant les $F4$. $F4$ est fonction de l'éventuelle présence d'un privilège en $P$, en $P^-$ et de la règle $R$. Entre $C$ et $C'$, lorsque $P$ change de valeur, $F4$ peut donc être modifié de la manière suivante :

1. Sur le couple ($P^-$,$P$) : la valeur de $P$ ayant changé, la définition de $F4(P^-,P)$ change aussi.
2. Sur le couple ($P$,$P^+$) : même chose.
3. Si $Test(P)$, anciennement à $A$, prend la valeur $R$ ou $Q_i$ : il est possible qu'il existe $P'$, un des successeurs de $P$, tel que l'évaluation de $Ch_{RQ}(P')$ entre dans le calcul de $F4(P'^-,P)$. Dans ce cas, $F4(P'^-,P')$ change de valeur entre $C$ et $C'$ et la modification de $Ch_{RQ}(P')$ doit être prise en considération dans le calcul de $F3$. On note $\Delta Ch$ la différence $(Ch_{RQ})_{C'}(P') - (Ch_{RQ})_C(P')$.

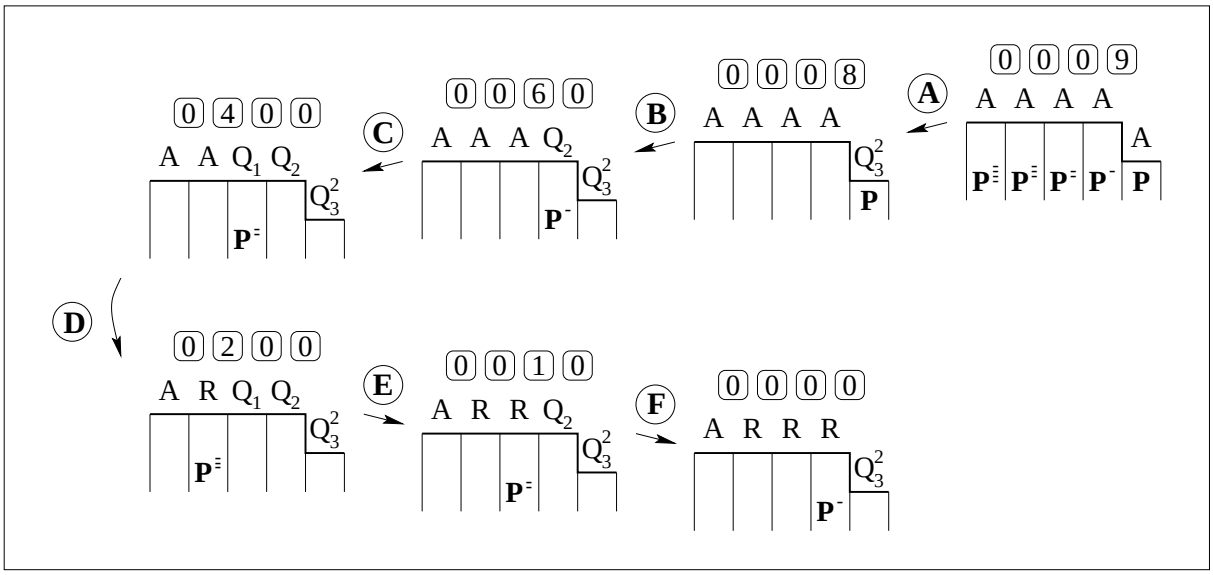
Au final, on a :

$$\Delta F3(T) = F4_{C'}(P^-,P) - F4_C(P^-,P) + F4_{C'}(P,P^+) - F4_C(P,P^+) + \Delta Ch$$

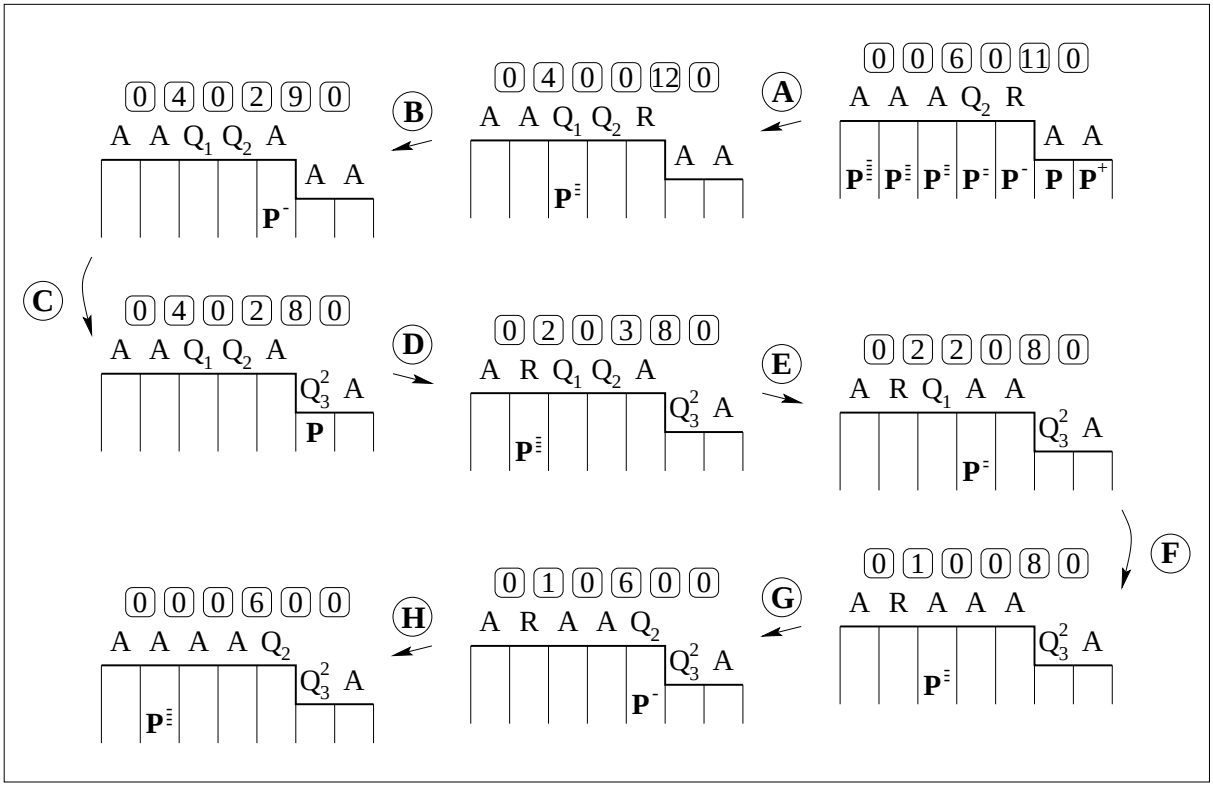
Pour exemple, nous traitons ici le cas particulier où la règle $R2$ est exécutée alors que ni $P$ ni $P^+$ n'ont le privilège.

Dans ce cadre, la règle $R2$ peut s'écrire

$$R2 : \neg Privilege(P) \wedge (A,\mathbf{A},Q_{i+1}) \Longrightarrow Test(P) \longleftarrow Q_i$$




(a) Exécution légitime




(b) Exécution quelconque

FIG. 5.8 – *Potentiel associé au test*

| | | |
|---|---|---|
| R1 : | $Privilege(P) \wedge (R,\mathbf{Q_k^j},?)$ | $\longrightarrow \begin{cases} Test(P) \longleftarrow A \\ Priv(P) \longleftarrow Priv(P^-)+1 \text{ si } P \text{ est le distingué.} \\ Priv(P) \longleftarrow Priv(P^-) \text{ sinon} \end{cases}$ |
| R2.a : | $Privilege(P) \wedge (A,\mathbf{A},?)$ | $\longrightarrow Test(P) \longleftarrow Q_k^j$ |
| R4.a : | $Privilege(P) \wedge (?,?_{\mathbf{R}}^{\mathbf{Q}},?)$ | $\longrightarrow Test(P) \longleftarrow A$ |
| R2.b : | $\neg Privilege(P) \wedge (A,\mathbf{A},Q_{i+1})$ | $\longrightarrow Test(P) \longleftarrow Q_i$ |
| R3.a : | $\neg Privilege(P) \wedge (A,\mathbf{A},Q_1)$ | $\longrightarrow Test(P) \longleftarrow R$ |
| R3.b : | $\neg Privilege(P) \wedge (R,\mathbf{Q_i},Q_{i+1})$ | $\longrightarrow Test(P) \longleftarrow R$ |
| R4.d : | $\neg Privilege(P) \wedge (?,?_{\mathbf{R}}^{\mathbf{Q}},A)$ | $\longrightarrow Test(P) \longleftarrow A$ |
| R4.f : | $\neg Privilege(P) \wedge (?,\mathbf{Q_i},Q_{i'})$ | $\longrightarrow Test(P) \longleftarrow A$ |

FIG. 5.9 – *Ré-écriture des règles*

Dans $C$, on a $(Test(P^-),\mathbf{Test(P)},Test(P^+)) = (A,\mathbf{A},Q_{i+1})$. D'où, d'après la définition 5.2.4, $F4(P^-,P)$ a pour valeur 0 et $F4(P,P^+)$ a pour valeur $k+2(i+1)-1$.

Après exécution de $R2$, on a $(Test(P^-),\mathbf{Test(P)},Test(P^+)) = (A,\mathbf{Q_i},Q_{i+1})$. D'ou $F4(P^-,P)$ a pour valeur $k+2i-1$ et $F4(P,P^+)$ a pour valeur 0.

Parallèlement à cela, $Test(P)$ est passé de $A$ a $Q_i$. Il est donc possible que cette transition ait augmenté la valeur de la chaîne d'un processeur $P'$ de 1. D'où $\Delta Ch = +1$

Au final, on a

$$\begin{array}{rcl} \Delta F3(T) & = & F4_{C'}(P^-,P) - F4_C(P^-,P) + F4_{C'}(P,P^+) - F4_C(P,P^+) + \Delta Ch \\ & = & (k-2i+1) - (0) + (0) - (k-2i-1) + 1 \\ & = & -1 \end{array}$$

et $\Delta F3$ est strictement négatif.

On procède de la même manière pour toutes les règles possibles et en fonction de la présence éventuelle du privilège en $P^-$, $P$ ou $P^+$. Les tableaux présentés figure 5.10 donnent un résumé complet des variations de $F3$.

Dans tous les cas, $\Delta F3$ est strictement négatif. □

### 5.2.4 Correction

**Survol de la preuve :** Pour prouver la correction, nous montrons que le nombre de privilèges n'augmente jamais au cours d'une exécution et qu'il en existe toujours au moins un. Nous établissons ensuite qu'il n'existe pas de configuration terminale. Tout cela nous permet de définir les configurations correctes.

Au final, nous montrons que l'ensemble des configurations légitimes est inclus dans l'ensemble des configurations correctes. Cela établit la correction.

#### Nombre de privilèges

**Lemme 5.2.7**
Dans toute configuration, il existe au moins un processeur privilégié.

La preuve de ce lemme est basée sur le même principe que celle du lemme 3.5.2, algo-

Le premier tableau récapitule les variations de $F4$ si ni $P^-$, ni $P$ ni $P^+$ n'ont de privilège :

| Règle | | $\mathbf{F4_{C_1}(P^-,P)}$ | $\mathbf{F4_{C_1}(P,P^+)}$ | $\mathbf{F4_{C_2}(P^-,P)}$ | $\mathbf{F4_{C_2}(P,P^+)}$ | $\mathbf{\Delta Ch}$ | $\mathbf{\Delta F3}$ |
|---|---|---|---|---|---|---|---|
| R2.b : | $(A,A,Q_{i+1}) \longrightarrow (A,Q_i,Q_{i+1})$ | 0 | $k+2(i+1)-1$ | $k+2(i)-1$ | 0 | +1 | -1 |
| R3.a : | $(A,A,Q_1) \longrightarrow (A,R,Q_1)$ | 0 | $k+2(1)-1$ | 0 | $k-(1)$ | +1 | -1 |
| R3.b : | $(R,Q_i,Q_{i+1}) \longrightarrow (R,R,Q_{i+1})$ | $k-(i)$ | 0 | 0 | $k-(i+1)$ | 0 | -1 |
| R4.d : | $(?,?^Q_R,A) \longrightarrow (?,A,A)$ | ? | $Ch_{RQ}(P)$ | $Ch_{RQ}(P^-)$ | 0 | 0 | -1 |
| R4.f : | $(?,Q_i,Q_{i'}) \longrightarrow (?,A,Q_{i'})$ | ? | $3k+Ch_{RQ}(P)$ | $Ch_{RQ}(P^-)$ | $k+2(i')-1$ | 0 | -2 |

Le deuxième tableau récapitule les variations de $F4$ lorsque $P^+$ a un privilège (et que $P^-$ et $P$ n'en ont pas). A noter, pour ce tableau comme pour les tableaux 3, 4, 6, 7 et 8, $\Delta Ch$ n'intervient pas. En effet, $\Delta Ch$ concerne un processeur $P'$ successeur de $P^+$. Or $P^+$ étant privilégié, la valeur de $P$ n'intervient pas dans le calcul de $Ch_{RQ}(P')$ (voir la définition de $Ch_{RQ}$, définition 5.2.2, page 87). Ce résultat s'étend au cas ou $P$ est lui-même privilégié.

| Règle | | $\mathbf{F4_{C_1}(P^-,P)}$ | $\mathbf{F4_{C_1}(P,P^+)}$ | $\mathbf{F4_{C_2}(P^-,P)}$ | $\mathbf{F4_{C_2}(P,P^+)}$ | $\mathbf{\Delta F3}$ |
|---|---|---|---|---|---|---|
| R2.b : | $(A,A,Q^j_k) \longrightarrow (A,Q_{k-1},Q^j_k)$ | 0 | $3k-1$ | $k+2(k-1)-1$ | 0 | -2 |
| R3.a : | $(A,A,Q_1) \longrightarrow (A,R,Q_1)$ | 0 | $3k+3$ | 0 | $3k+Ch_{RQ}(P^+)$ | -1 |
| R3.b : | $(R,Q_{k-1},Q_k) \longrightarrow (R,R,Q_k)$ | $k-(k-1)$ | 0 | 0 | 0 | -1 |
| R3.b : | $(R,Q_i,Q_{i+1}) \longrightarrow (R,R,Q_i)$ | $k-(i)$ | $3k+Ch_{RQ}(P^+)$ | 0 | $3k+Ch_{RQ}(P^+)$ | -k+i |
| R4.d : | $(?,?^Q_R,A) \longrightarrow (?,A,A)$ | ? | $3k+Ch_{RQ}(P)$ | $Ch_{RQ}(P^-)$ | $3k$ | -1 |
| R4.f : | $(?,Q_i,Q_{i'}) \longrightarrow (?,A,Q_{i'})$ | ? | $3k+Ch_{RQ}(P^+)$ | $Ch_{RQ}(P^-)$ | $3k+Ch_{RQ}(P^+)$ | -1 |

Le troisième tableau récapitule les variations de $F4$ si seul $P$ a un privilège.

| Règle | | $\mathbf{F4_{C_1}(P^-,P)}$ | $\mathbf{F4_{C_1}(P,P^+)}$ | $\mathbf{F4_{C_2}(P^-,P)}$ | $\mathbf{F4_{C_2}(P,P^+)}$ | $\mathbf{\Delta F3}$ |
|---|---|---|---|---|---|---|
| R2.a | $(A,A,?) \rightarrow (A,Q^j_k,?)$ | $3k$ | 0 | $3k-1$ | 0 | -1 |
| R4.a | $(A,Q_1,?) \rightarrow (A,A,?)$ | $3k+3$ | 0 | $3k$ | 0 | -3 |
| R4.a | $(?,?^{Q'}_R,?) \rightarrow (?,A,?)$ | $3k+Ch_{RQ}(P)$ | ? | $3k+Ch_{RQ}(P^-)$ | 0 | -2 |

Le quatrième tableau récapitule les variations de $F4$ si $P$ et $P^+$ ont un privilège.

| Règle | | $\mathbf{F4_{C_1}(P^-,P)}$ | $\mathbf{F4_{C_1}(P,P^+)}$ | $\mathbf{F4_{C_2}(P^-,P)}$ | $\mathbf{F4_{C_2}(P,P^+)}$ | $\mathbf{\Delta F3}$ |
|---|---|---|---|---|---|---|
| R2.a | $(A,A,A) \rightarrow (A,\tilde{Q},A)$ | $3k$ | 2 | $3k-1$ | 0 | -3 |
| R2.a | $(A,A,?^{Q'}_R) \rightarrow (A,\tilde{Q},?^{Q'}_R)$ | $3k$ | 3 | $3k-1$ | 3 | -1 |
| R2.a | $(A,A,\tilde{Q}) \rightarrow (A,\tilde{Q},\tilde{Q})$ | $3k$ | 1 | $3k-1$ | 1 | -1 |
| R4.a | $(A,Q_1,?) \rightarrow (A,A,?)$ | $3k+3$ | 2 | $3k$ | 3 | -2 |
| R4.a | $(?,?^{Q'}_R,A) \rightarrow (?,A,A)$ | $3k+Ch_{RQ}(P)$ | 2 | $3k+Ch_{RQ}(P^-)$ | 2 | -1 |
| R4.a | $(?,?^{Q'}_R,?^{Q'}_R) \rightarrow (?,A,?^{Q'}_R)$ | $3k+Ch_{RQ}(P)$ | 3 | $3k+Ch_{RQ}(P^-)$ | 3 | -1 |
| R4.a | $(?,?^{Q'}_R,\tilde{Q}) \rightarrow (?,A,\tilde{Q})$ | $3k+Ch_{RQ}(P)$ | 3 | $3k+Ch_{RQ}(P^-)$ | 1 | -3 |

FIG. 5.10 – *Décroissance de $F3$*

Les tableaux 5, 6, 7 et 8 sont à rapprocher respectivement des tableaux 1, 2, 3 et 4. En effet, entre les premiers et les seconds, seule change la présence d'un privilège en $P^-$.
Le cinquième tableau récapitule les variations de $F4$ si seul $P^-$ a un privilège.

| Règle | | $\mathbf{F4_{C_1}(P^-,P)}$ | $\mathbf{F4_{C_1}(P,P^+)}$ | $\mathbf{F4_{C_2}(P^-,P)}$ | $\mathbf{F4_{C_2}(P,P^+)}$ | **ΔCh** | **ΔF3** |
|---|---|---|---|---|---|---|---|
| R2.b : | $(A,A,Q_{i+1}) \longrightarrow (A,Q_i,Q_{i+1})$ | 0 | $k+2(i+1)-1$ | 0 | 0 | +1 | -k-2i |
| R3.a : | $(A,A,Q_1) \longrightarrow (A,R,Q_1)$ | 0 | $k+2(1)-1$ | 0 | $k-(1)$ | +1 | -1 |
| R3.b : | $(R,Q_i,Q_{i+1}) \longrightarrow (R,R,Q_{i+1})$ | $k-(i)$ | 0 | 0 | $k-(i+1)$ | 0 | -1 |
| R4.d : | $(?,?^Q_R,A) \longrightarrow (?,A,A)$ | ? | $Ch_{RQ}(P)$ | 0 | 0 | 0 | -1 |
| R4.f : | $(?,Q_i,Q_{i'}) \longrightarrow (?,A,Q_{i'})$ | ? | $3k+Ch_{RQ}(P)$ | 0 | $k+2(i')-1$ | 0 | -2 |

Le sixième tableau récapitule les variations de $F4$ si $P^-$ et $P^+$ ont un privilège.

| Règle | | $\mathbf{F4_{C_1}(P^-,P)}$ | $\mathbf{F4_{C_1}(P,P^+)}$ | $\mathbf{F4_{C_2}(P^-,P)}$ | $\mathbf{F4_{C_2}(P,P^+)}$ | **ΔF3** |
|---|---|---|---|---|---|---|
| R2.b : | $(A,A,Q^j_k) \longrightarrow (A,Q_{k-1},Q^j_k)$ | 0 | $3k-1$ | $k+2(k-1)-1$ | 0 | -2 |
| R3.a : | $(A,A,Q_1) \longrightarrow (A,R,Q_1)$ | 0 | $3k+3$ | 0 | $3k+Ch_{RQ}(P^+)$ | -1 |
| R3.b : | $(R,Q_{k-1},Q_k) \longrightarrow (R,R,Q_k)$ | $k-(k-1)$ | 0 | 0 | 0 | -1 |
| R3.b : | $(R,Q_i,Q_{i+1}) \longrightarrow (R,R,Q_i)$ | $k-(i)$ | $3k+Ch_{RQ}(P^+)$ | 0 | $3k+Ch_{RQ}(P^+)$ | -k+i |
| R4.d : | $(?,?^Q_R,A) \longrightarrow (?,A,A)$ | ? | $3k+Ch_{RQ}(P)$ | $Ch_{RQ}(P^-)$ | $3k$ | -1 |
| R4.f : | $(?,Q_i,Q_{i'}) \longrightarrow (?,A,Q_{i'})$ | ? | $3k+Ch_{RQ}(P^+)$ | $Ch_{RQ}(P^-)$ | $3k+Ch_{RQ}(P^+)$ | -1 |

Le septième tableau récapitule les variations de $F4$ si $P^-$ et $P$ ont un privilège.

| Règle | | $\mathbf{F4_{C_1}(P^-,P)}$ | $\mathbf{F4_{C_1}(P,P^+)}$ | $\mathbf{F4_{C_2}(P^-,P)}$ | $\mathbf{F4_{C_2}(P,P^+)}$ | **ΔF3** |
|---|---|---|---|---|---|---|
| R2.a | $(A,A,?) \rightarrow (A,Q^j_k,?)$ | 2 | 0 | 1 | 0 | -1 |
| R4.a | $(?,?^{Q'}_R,?) \rightarrow (?,A,?)$ | 3 | ? | 2 | 0 | -1 |

Le huitième et dernier tableau (ouf) récapitule les variations de $F4$ si $P^-$, $P$ et $P^+$ ont tous un privilège.

| Règle | | $\mathbf{F4_{C_1}(P^-,P)}$ | $\mathbf{F4_{C_1}(P,P^+)}$ | $\mathbf{F4_{C_2}(P^-,P)}$ | $\mathbf{F4_{C_2}(P,P^+)}$ | **ΔF3** |
|---|---|---|---|---|---|---|
| R2.a | $(A,A,A) \rightarrow (A,\tilde{Q},A)$ | 2 | 2 | 1 | 0 | -3 |
| R2.a | $(A,A,?^{Q'}_R) \rightarrow (A,\tilde{Q},?^{Q'}_R)$ | 2 | 3 | 1 | 3 | -1 |
| R2.a | $(A,A,\tilde{Q}) \rightarrow (A,\tilde{Q},\tilde{Q})$ | 2 | 1 | 1 | 1 | -1 |
| R4.a | $(?,?^{Q'}_R,A) \rightarrow (?,A,A)$ | 3 | 2 | 2 | 2 | -1 |
| R4.a | $(?,?^{Q'}_R,?^{Q'}_R) \rightarrow (?,A,?^{Q'}_R)$ | 3 | 3 | 2 | 3 | -1 |
| R4.a | $(?,?^{Q'}_R,\tilde{Q}) \rightarrow (?,A,\tilde{Q})$ | 3 | 3 | 2 | 1 | -3 |

Fig 5.10 (suite) - Décroissance de $F3$

rithme de Dijkstra.

**Preuve :** Dans une configuration donnée, l'existence de privilèges dépend des valeurs de la variable $Priv$.

1. Si la variable $Priv$ du processeur distingué est la même que celle de son prédécesseur, le distingué a un privilège.
2. Sinon, il existe au moins un processeur dont la variable $Priv$ n'a pas la même valeur que celle de son prédécesseur. Ce processeur possède un privilège.

□

**Lemme 5.2.8**
Au cours d'une exécution, le nombre de privilèges n'augmente pas.

Cette preuve est basée sur le même principe que celle du lemme 3.5.4, algorithme de Dijkstra.

**Preuve :** Dans une configuration donnée, l'existence de privilèges dépend des valeurs de la variable $Priv$. Or seule la règle $R1$ modifie la valeur de $Priv$ et seul un processeur possédant un privilège peut appliquer $R1$. Ce faisant, il perd son privilège. Même si $P^+$ en acquiert un, le nombre global de privilèges n'augmente pas.

□

**Configuration terminale**

**Lemme 5.2.9**
Soit $k$ une constante strictement inférieure a $\sqrt{n}-1$. Alors dans toute configuration n'ayant pas plus de $k$ processeurs corrompus, il existe un processeur privilégié n'ayant pas de privilèges parmi ses $k$ prédécesseurs.

**Preuve :** Etant donné une configuration $C$ dont $i$ processeurs sont corrompus, considérons $\mathcal{N}$, l'ensemble des processeurs n'ayant pas de privilégiés parmi leur $k$ prédécesseurs. Montrons par récurrence sur $i$ que $\mathcal{N}$ a pour cardinal $n-k-i(k+1)$.

1. Si $i = 0$ : $C$ est une configuration légitime. Il n'y a donc qu'un seul privilège dans $C$ et seuls ses $k$ successeurs ne sont pas dans $\mathcal{N}$. On a donc $Cardinal(\mathcal{N}(C)) = n-k$
2. Supposons la propriété vraie pour une configuration dont $i-1$ processeurs sont corrompus.
3. Montrons la propriété pour $i$ : $C$ est une configuration obtenue en corrompant $k$ processeur $\{P_j\}$ d'une configuration légitime $L$. Considérons $C'$, la configuration obtenue à partir de $C$ en donnant à $P_1$ la valeur qu'il a dans $L$. Il y a donc $k-1$ processeurs corrompus dans $C'$ et $Cardinal(\mathcal{N}(C')) = n-k-(i-1)(k+1)$. Dans $C$, de par la corruption de $P_1$, $P_1$ et $P_1^+$ sont privilégiés et donc $P_1^{2+}$, $P_1^{3+}$, ..., $P_1^{(k+1)+}$ ne peuvent pas être dans $\mathcal{N}(C)$, soit un total de $k+1$ processeurs qui sont peut-être dans $\mathcal{N}(C')$ mais pas dans $\mathcal{N}(C)$. On a donc

$$\begin{aligned} Cardinal(\mathcal{N}(C)) &\geq Cardinal(\mathcal{N}(C')) - k - 1 \\ &\geq n - k - i(k+1) \end{aligned}$$

Donc, pour une configuration $C$ dont $k$ processeurs sont corrompus, on a $Cardinal(\mathcal{N}(C)) \geq n - k - k(k+1)$. Or, on veut que $\mathcal{N}$ soit non vide, c'est à dire :

$$\begin{array}{lcl} n-k-k(k+1) & > & 0 \\ n+1-(k^2+2k+1) & > & 0 \\ (\sqrt{n+1})^2-(k+1)^2 & > & 0 \end{array}$$

Comme les deux termes de l'équation sont positifs, on obtient :

$$k < \sqrt{n+1} - 11$$

D'où, si $k$ est strictement inferieur à $\sqrt{n}-1$, $\mathcal{N}(C)$ est non vide et il existe dans $C$ un processeur n'ayant pas de privilège parmi ses $k$ prédécesseurs. □

**Lemme 5.2.10**
Il n'existe pas de configuration terminale.

**Preuve :** (par l'absurde). Soit $C$ une configuration et $P$ un processeur privilégié n'ayant pas de privilégié parmi ses $k$ prédécésseurs. Supposons que $C$ soit terminale.

1. Si $Autorisation(P)$ est vrai, $P$ peut appliquer $R1$.
2. Si $Question(P)$ est vrai, $P$ peut appliquer $R2$.
3. Si $Probleme(P)$ est vrai, $P$ peut appliquer $R4$.

Donc, comme $C$ est terminale, $TestEnCour(P)$ est vrai et $Test(P) = Q_k^j$. Soit $P^{i-}$ le premier prédécesseur de $P$ ne vérifiant pas $(Test(P^{i-},P^{(i-1)-})) = (Q_i,Q_{i+1})$.

1. Si $Test(P^{i-}) = R$, alors $P^{(i-1)-}$ peut appliquer la règle $R3$.
2. Si $Test(P^{i-}) = A$ et $Test(P^{(i+1)-}) = A$, alors soit $P^{i-}$ peut appliquer la règle $R2$.
3. Si $Test(P^{i-}) = A$ et $Test(P^{(i+1)-}) \neq A$, alors $P^{(i+1)-}$) a son prédicat $Probleme$ a vrai et il peut appliquer $R4$.
4. Si $Test(P^{i-}) = Q_{i'}$, alors $P^{i-}$ peut appliquer la règle $R4$.

Dans tous les cas, un processeur peut agir et l'hypothèse selon laquelle $C$ est terminale était donc fausse. □

### Configurations correctes

**Lemme 5.2.11 : Configuration correcte**
L'ensemble des configurations correctes est l'ensemble des configurations dans lesquelles un unique processeur $P$ est privilègié.

**Preuve :** Une configuration contenant plus d'un privilège ne vérifie pas la spécification de l'exclusion mutuelle. Les configurations sans privilège n'existant pas (lemme 5.2.7) seules les configurations dans lesquelles un unique privilège est présent peuvent être correctes. Soit $L$ une configuration n'ayant qu'un seul processeur privilégié. Montrons que toute exécution $\mathcal{E}$ dont $L$ est configuration initiale vérifie la spécification de l'exclusion mutuelle.

Le nombre de privilèges n'augmentant jamais lors d'une exécution (lemme 5.2.8), toute configuration de $\mathcal{E}$ ne comporte qu'un seul privilège.

D'autre part, la fonction $F3$ est positive strictement décroissante sur toute transition n'impliquant pas de mouvement de privilège (lemme 5.2.6). Toute exécution étant infinie (il n'existe pas de configuration terminale, (lemme 5.2.10), $\mathcal{E}$ contient forcément une infinité de transmission de privilèges. La circulation du privilège se faisant toujours dans le même sens, (lemme 5.2.12), tous les processeurs le reçoivent infiniment souvent et la spécification de l'exclusion mutuelle est vérifiée.

□

**Remarque 5.2.12**
Comme pour l'algorithme de Dijkstra, lorsqu'un processeur $P$ applique la règle $R1$, il perd son privilège. Si $P^+$ n'a pas de privilège, il en acquiert un et la transmission de privilèges se fait toujours d'un processeur vers son successeur.

**Lemme 5.2.13**
L'algorithme présenté figure 5.5, page 85 vérifie la propriété de correction.

**Preuve :** Dans une configuration légitime, un seul processeur est privilégié (definition 5.2.3) donc toute configuration légitime est correcte. Cela établit la convergence.

□

### 5.2.5 Anti-corruption

**Définition 5.2.14**
Soit $C$ une configuration illégitime obtenue à partir de la configuration légitime $L$. Soit $P_0$ le processeur possédant un privilège dans $L$. On distingue dans $C$ trois types de privilèges :

1. $P$ a un privilège *faux* si $Priv(P^-)$ est corrompu.
2. $P$ a un privilège *correcteur* si $Priv(P^-)$ n'est pas corrompu.
3. Le *vrai* privilège est :
   (a) Le privilège détenu par $P_0$ si $P_0^-$ n'est pas corrompu.
   (b) Le plus proche privilège correcteur précédent $P_0$ sur l'anneau si $P_0^-$ est corrompu.

Un exemple illustrant la nature des privilèges est donné figure 5.10. Dans la configuration légitime $L$, un seul processeur a un privilège. Dans la configuration $C_1$ obtenue en corrompant deux processeurs de $L$, trois processeurs sont privilégiés. Celui qui avait un privilège dans $L$ a le vrai privilége, celui dont le prédécesseur est corrompu a un faux privilège et celui dont le prédécesseur est non corrompu a un privilège correcteur. Dans la configuration $C_2$, le processeur dont le prédécesseur est corrompu a un faux privilège et celui dont le prédécesseur est non corrompu a un privilège correcteur. C'est également le vrai privilège, puisque le privilègié dans $L$ est corrompu dans $C_2$.

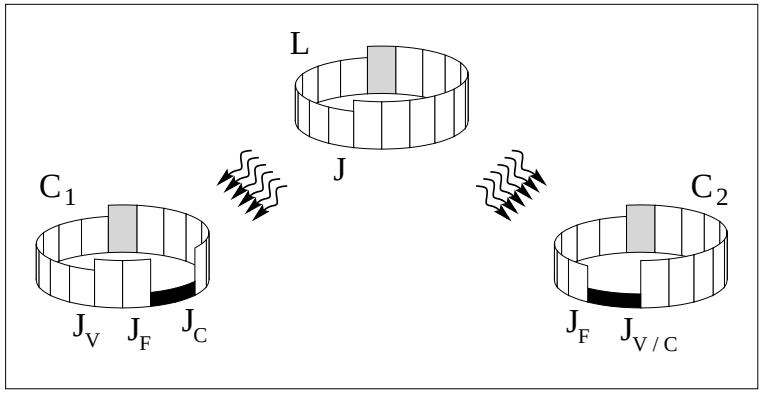



FIG. 5.10 – *Nature des privilèges*

**Lemme 5.2.15**
Soit $C$ une configuration. Soit $P$ un processeur non corrompu possédant un privilège. $P$ ne peut transmettre son privilège que s'il s'agit d'un privilège vrai ou d'un correcteur.

La preuve de ce lemme est basée sur le principe suivant : si dans une configuration donnée, la variable d'un processeur a une certaine valeur, c'est soit que le processeur en question a appliqué une règle le permettant, soit que la valeur de la variable a la même valeur depuis le début de l'exécution. Dans le premier cas, il est possible de faire des déductions sur l'histoire de l'exécution. Par exemple, s'il est certain qu'un processeur a eu à un moment donné sa variable $Test$ sur $A$ et qu'il a actuellement sa variable $Test$ sur $R$, il a nécessairement appliqué auparavant la règle $R3$, et encore auparavant la règle $R2$. De proche en proche, on peut ainsi déduire une bonne partie des transitions permettant de conduire à une configuration.

**Preuve :** Soit $C$ une configuration et $P$ un processeur privilégié. $P$ peut transmettre un privilège, c'est à dire qu'il peut appliquer la règle $R1$. On a donc $Test(P^-,P) = (R,Q_k^j)$ avec $j = Priv(P^-)$.

1. Si $Test(P)$ n'a pas changé de valeur depuis le début de l'exécution : la valeur du privilège de $P$ est correcte. En effet, $j$ a la même valeur que dans la configuration initiale $C_0$. $P$ n'étant pas corrompu, $j$ a la même valeur que dans la configuration légitime $L$ dont est issu $C_0$. C'est donc que $P^-$ n'a pas une valeur corrompue et $P$ est un privilège vrai ou correcteur.
2. Si $Test(P)$ a changé de valeur : dans $C$, on a $Test(P^-,P) = (R,Q_k^j)$. Donc $P$ a appliqué $R2$ : $(A,\mathbf{A},?) \Longrightarrow (A,\mathbf{Q_k^j},?)$ (car c'est la seule règle permettant à $Test(P)$ de prendre la valeur $Q_k^j$).
   On sait aussi que $P^-$ a appliqué $R3$ : $(R,\mathbf{Q_{k-1}},Q_k^j) \Longrightarrow (R,\mathbf{Q_{k-1}},Q_k^j)$ (car $R3$ est la seule règle permettant a $Test(P^-)$ de prendre la valeur $R$) et qu'auparavant, $P^-$ avait appliqué $R2$ : $(A,\mathbf{A},Q_k^j) \Longrightarrow (A,\mathbf{Q_{k-1}},Q_k^j)$ (car c'est la seule règle qui peut donner à $Test(P^-)$ la valeur $Q_{k-1}$, valeur qu'il doit avoir avant d'appliquer $R3$).

De la même manière, pour que $P^-$ puisse appliquer $R2$, il faut que $Test(P^{--})$ vaille $A$ et pour que $P^-$ puisse appliquer $R3$, il faut que $Test(P^{--})$ vaille $R$. Donc $P^{--}$ a successivement appliqué les règles $R2$ puis $R3$.

De proche en proche, les $k-1$ prédécesseurs de $P$ ont appliqué $R2$ puis $R3$. $P^{(k-1)-}$ a appliqué $R2$ : $(A,\mathbf{A},Q_2) \Longrightarrow (A,\mathbf{Q_1},2)$, puis $R3$ : $(R,\mathbf{Q_1},Q_2) \Longrightarrow (R,R,Q_2)$. Par contre, pour passer de la valeur $A$ à $R$, $P^{k-}$ n'a pu qu'appliquer $R3$.

En résumé, si $P$ a l'autorisation de transmettre son privilège, c'est que ses $k$ prédécesseurs ont eu à appliquer $R3$. Or $R3$ n'est applicable que par un processeur n'ayant pas de privilège. Donc $P$ est assuré que ses $k$ prédécesseurs n'ont pas de privilège. Le privilège de $P$ n'est donc pas un faux privilège.

Dans un cas comme dans l'autre, le privilège de $P$ est un vrai privilège, ou un privilège correcteur. □

**Définition 5.2.16**
$\delta(A,B)$, la distance de Dirac, est définie comme valant 0 si $A$ et $B$ sont identiques, 1 sinon.

$$\delta(A,B) = \begin{cases} 0 & \text{si } A = B \\ 1 & \text{sinon} \end{cases}$$

**Lemme 5.2.17**
Soit $T = (C_1, \stackrel{(P,R1)}{\longrightarrow}, C_2)$ une transition. Si le privilège de $P$ est un vrai privilège, alors $T$ n'est pas une transition corruptrice.

**Preuve :** Soit $C_1'$ une configuration correcte réalisant la distance entre $C_1$ et l'ensemble des configurations correctes $\mathcal{CC}$. $P$ a un privilège vrai, $P^-$ n'est donc pas un processeur corrompu et $Priv(P^-)$ a la même valeur dans $C_1$ que dans $C_1'$.

Soit $C_2'$ la configuration obtenue à partir de $C_1'$ en donnant $Priv(P)$ la valeur que $Priv(P^-)$ a dans $C_1'$. $C_2'$ est une configuration correcte. En effet, si dans $C_1'$, $P$ n'a pas de privilège, alors $C_2' = C_1'$. Si $P$ a un unique privilège, lui donner la valeur de son prédécsseur ne fait que transmettre le privilège à $P^+$.

Evaluons maintenant la distance entre $C_2'$ et $C_2$. Pour tous les processeurs $P'$ différents de $P$, on a

$$Priv_{C_1}(P') = Priv_{C_1'}(P') \Longrightarrow Priv_{C_2}(P') = Priv_{C_2'}(P')$$

$$Priv_{C_1}(P') \neq Priv_{C_1'}(P') \Longrightarrow Priv_{C_2}(P') \neq Priv_{C_2'}(P')$$

Pour $P$, on a

$$\begin{array}{rcll} Priv_{C_2}(P) & = & Priv_{C_1}(P^-) & \text{car dans } C_2,\ P \text{ a transmis son privilège} \\ & = & Priv_{C_1'}(P^-) & \text{car } P^- \text{ n'est pas corrompu} \\ & = & Priv_{C_2'}(P) & \text{par construction de } C_2' \end{array}$$

On a donc $\delta(Priv_{C_2}(P),Priv_{C_2'}(P)) = 0$.

Au final, la distance de hamming entre deux configurations étant le nombre de processeurs n'ayant pas les mêmes valeurs dans les deux configurations, on a :

$$\begin{array}{rcl} Dist(C_1,C'_1) & = & \delta(Priv_{C_1}(P),Priv_{C'_1}(P)) + \sum_{P'\neq P}\{\delta(Priv_{C_1}(P'),Priv_{C'_1}(P'))\} \\ & = & \delta(Priv_{C_1}(P),Priv_{C'_1}(P)) + \sum_{P'\neq P}\{\delta(Priv_{C_2}(P'),Priv_{C'_2}(P'))\} \\ & \geq & \delta(Priv_{C_2}(P),Priv_{C'_2}(P)) + \sum_{P'\neq P}\{\delta(Priv_{C_2}(P'),Priv_{C'_2}(P'))\} \end{array}$$

D'ou $Dist(C_1,C'_1) \geq Dist(C_2,C'_2)$ et $T$ n'est pas une transition corruptrice. □

**Lemme 5.2.18**
Soit $T = (C_1, \stackrel{(P,R1)}{\longrightarrow}, C_2)$ une transition. Si le privilège de $P$ est un privilège correcteur, alors $T$ est une transition correctrice.

**Preuve :** Soit $C'_1$ une configuration correcte réalisant la distance entre $C_1$ et $\mathcal{CC}$. $P$ a un privilège correcteur, $P^-$ n'est donc pas un processeur corrompu. Par contre, $P$ est corrompu. $Priv(P^-)$ a la même valeur dans $C_1$ que dans $C'_1$.

Soit $C'_2$ la configuration obtenue à partir de $C'_1$ en donnant $Priv(P)$ la valeur que $Priv(P^-)$ a dans $C'_1$. $C'_2$ est une configuration correcte (mêmes arguments que la preuve précédente).

Evaluons maintenant la distance entre $C'_2$ et $C_2$. Pour tous les processeurs $P'$ differents de $P$, on a

$$\delta(Priv_{C_1}(P'),Priv_{C'_1}(P')) = \delta(Priv_{C_2}(P'),Priv_{C'_2}(P'))$$

Pour $P$, on a $Priv_{C_2}(P) = Priv_{C'_2}(P)$ et $\delta(Priv_{C_2}(P),Priv_{C'_2}(P)) = 0$. D'un autre côté, $P$ etant corrompu dans $C_1$, on a $Priv_{C_1}(P) \neq Priv_{C'_1}(P)$.

Au final, la distance de hamming entre deux configurations étant le nombre de processeurs n'ayant pas les mêmes valeurs dans les deux configurations, on a :

$$\begin{array}{rcll} Dist(C_1,C'_1) & = & \delta(Priv_{C_1}(P),Priv_{C'_1}(P)) & + \sum_{P'\neq P}\{\delta(Priv_{C_1}(P'),Priv_{C'_1}(P'))\} \\ & = & 1 & + \sum_{P'\neq P}\{\delta(Priv_{C_2}(P'),Priv_{C'_2}(P'))\} \\ & > & 0 & + \sum_{P'\neq P}\{\delta(Priv_{C_2}(P'),Priv_{C'_2}(P'))\} \\ & > & Dist(C_2,C'_2) & \end{array}$$

D'ou $Dist(C_1,C'_1) > Dist(C_2,C'_2)$ et $T$ est une transition correctrice. □

**Lemme 5.2.19**
Soit $\mathcal{E}$ une exécution de l'algorithme présenté figure 5.5, page 85. Alors $\mathcal{E}$ ne contient aucune transition corruptrice.

**Preuve :** Soit $T = (C_1, \stackrel{(P,R)}{\longrightarrow}, C_2)$ une transition de $\mathcal{E}$. Si $R$ n'est pas $R1$, la variable de sortie $Priv$ n'est pas modifiée et la transition $T$ n'est pas corruptrice.

Si $R = R1$, $P$ dispose soit d'un privilège vrai, soit d'un privilège correcteur (lemme 5.2.15).

1. Si $P$ dispose d'un privilège vrai : la transition $T$ n'est pas corruptrice (lemme 5.2.17)
2. Si $P$ dispose d'un privilège vrai : la transition $T$ est correctrice (lemme 5.2.18)

Dans tous les cas, $T$ n'est pas une transition corruptrice. □

### 5.2.6 $K$-convergence

**Définition 5.2.20**
Soit $C$ une configuration. Soit $P$ le vrai privilège de $C$ et $P'$ le plus proche prédécesseur de $P$ possédant un privilège. $F5$ est la distance entre $C$ et l'ensemble des configurations correctes plus la distance (sur l'anneau) entre $P$ et $P'$ (distance avec le respect de l'orientation).

$$F5(C) = Dist(C,\mathcal{CC}) + Dist(P,P')$$

**Lemme 5.2.21**
Sur toute transition impliquant un mouvement de privilège, $F5$ décroît. Sur les autres transitions, $F5$ est inchangée.

**Preuve :** Soit $T = (C_1, \stackrel{(P,R)}{\longrightarrow}, C_2)$ une transition.

1. Si $R = R1 : P$ a soit un vrai privilège, soit un privilège correcteur (lemme 5.2.15) :
   (a) Si $P$ a un vrai privilège, la distance entre $P$ et $P'$ décroît et $Dist(C,\mathcal{CC})$ n'augmente pas (lemme 5.2.17)
   (b) Si $P$ a un privilège correcteur, $Dist(C,\mathcal{CC})$ decroit (lemme 5.2.18).

   $F5$ décroît donc sur toutes les transitions impliquant des mouvements de privilège.
2. Si $R \neq R1$ : entre $C_1$ et $C_2$, aucun privilége n'a changé de place, aucune des variables $Priv(P)$ n'a été modifiée. La distance entre $P$ et $P'$ ne change donc pas, pas plus que la distance entre $C_1$ et l'ensemble des configurations correctes. D'ou $F5$ ne change pas de valeur sur les transitions n'impliquant de mouvement de privilège.

□

**Lemme 5.2.22**
Soit $T = (C_1, \stackrel{(P,R1)}{\longrightarrow}, C_2)$ une transition. Alors $\Delta F3(T)$ est borné par $6k + 3$.

**Preuve :** Quand $R1$ est appliqué, $F4$ peut changer de valeur sur les couples $(P^-,P)$ et $(P,P^+)$ ($Test(P)$ prenant la valeur $A$, l'application de $R1$ par $P$ ne peut pas augmenter la taille de la chaîne $Ch_{RQ}$ d'un des successeurs de $P^+$).

A partir de la définition de $F4$ (définition 5.2.4), on observe que $F4(P,P^+)$ est majoré par $3k + 3$ (car $Ch_{RQ}(P) = 0$ et que $Ch_{RQ}(P^+) \leq 1$). D'un autre côté, $F4(P^-,P)$ est majoré par $Ch_{P^-}$.

Dans une configuration légitime, au plus $k$ processeurs (ceux qui sont impliqués dans un $Test$) peuvent avoir leur variable $Test$ à $R$ ou $Q_i$. Donc, pour $P$ quelconque, on a $Ch_P \leq k$.

Dans une configuration obtenue à partir d'une configuration légitime en corrompant $k$ processeurs, au plus $2k$ processeurs (ceux qui s'étaient impliqués dans un $Test$ dans la configuration $L$ plus ceux qui sont corrompus) peuvent avoir leur variable $Test$ à $R$ ou $Q_i$. Donc, pour $P$ quelconque, on a $Ch_P \leq 2k$

Au cours d'une exécution, seule une chaîne dont le dernier élement est $Q_i$ peut voir sa taille augmenter. Ce faisant, le dernier élément de la chaine devient $Q_{i-1}$ (ou $R$ si $i = 1$). La taille de la chaîne augmente donc au plus de $i$. Dans le pire des cas, un processeur peut donc avoir une chaîne de taille $3k$.

Au final, $F4(P^-,P)$ est majoré par $3k$ et $\Delta F3(T) \leq 6k + 3$. □

**Lemme 5.2.23**
Soit $F6$ la fonction définie par $F6(C) = F5(C) \times (6k + 4) + F3(C)$. Alors $F6$ est une fonction positive et strictement décroissante sur toute transition.

**Preuve :** Soit $T = (C_1, \xrightarrow{R}, C_2)$ une transition. Evaluons $\Delta F6(T)$

1. Si $R = R1$, alors $\Delta F5(T) = -1$ (lemme 5.2.21) et $\Delta F3(T) = 6k + 3$ (lemme 5.2.22). D'où $\Delta F6(T) = -1$
2. Si $R = R1$, alors $\Delta F5(T) = 0$ (lemme 5.2.21) et $\Delta F3(T) = -1$ (lemme 5.2.6). D'où $\Delta F6(T) = -1$

Au final, $F6$ est strictement décroissante. □

**Lemme 5.2.24**
L'algorithme présenté figure 5.5, page 85 vérifie la propriété de convergence.

**Preuve :** $F6$ est une fonction de convergence (lemme 5.2.23. De plus, il n'existe pas de configuration terminale (lemme 5.2.10). Ces deux propriétés suffisent à assurer la convergence (lemme 2.3.4, page 47). □

**Complexité** La fonction de convergence nous donne une borne supérieure sur le temps de convergence.

**Lemme 5.2.25**
Le temps de convergence de l'algorithme présenté figure 5.5, page 85 est de $O(kn)$.

**Preuve :** Soit $C_0$ la configuration initiale d'une exécution. Le nombre de corrompus présents dans $C_0$ ne peut excéder $k$ (d'où $Dist(C_0,\mathcal{CC}) \leq k$) et la distance entre $P$ et $P'$, son plus proche prédécesseur possédant un privilège, est inférieure à $n$. $F5(C_0)$ est donc bornée par $n + k$.

D'autre part, $F3(C_0)$ est borné par $3k$ (lemme 5.2.22).

Au final, on a $F6(C_0) \leq (n+k)(6k+3)+3k$ et l'algorithme converge en $O(kn)$ transitions. □

### 5.2.7 Bilan

Tous les éléments nécessaires à la preuve du théorème 3.2.1 sont maintenant réunis.

**Théorème 5.2.1**
Soit $k$ une constante strictement inférieure à $\sqrt{n}-1$. L'algorithme d'exclusion mutuelle présenté figure 5.5, page 85 est $k$-stabilisant et anti-corruption. De plus, si $n$ est la taille de l'anneau, il converge en $O(kn)$ transitions (complexité au pire) et nécessite $O(log(k))$ bits par processeur.

**Preuve :** L'ensemble de configurations légitimes $\mathcal{L}$ (définie au 5.2.3) vérifie la correction (lemme 5.2.13). La $k$-convergence vers une configuration correcte est assurée (lemme 5.2.24) tout comme l'anti-corruption (lemme 5.2.19). De plus, la phase de stabilisation de l'ordre de $kn$ étapes (lemme 5.2.25). Chaque processeur utilisant deux variables, l'un de taille $k$ et l'autre de taille $2k$, l'espace mémoire nécéssaire est de l'ordre de $log(k)$. D'où le système est $k$-stabilisant et anti-corruption pour l'exclusion mutuelle, son temps de convergence est $O(kn)$ et sa complexité en espace est $O(log(k))$. □

# Chapitre 6

# $k$-stabilisation rapide, anti-corruption et auto-stabilisante

L'algorithme que nous avons de décrire et de (longuement) prouver dans le chapitre précédent -nous l'appellerons dorénavant algorithme de base- est $k$-stabilisant, pour $k$ fixé strictement inférieur à $\sqrt{n} - 1$. En revanche, il n'est pas auto-stabilisant. En effet, si le nombre de fautes est plus grand que $k$, il peut ne plus converger.

Dans ce chapitre, nous présentons un nouvel algorithme, amélioration de l'algorithme de base, qui tout en conservant les propriétés d'anti-corruption et de $k$-stabilisation rapide, assure en outre la convergence en cas de corruption totale et est donc auto-stabilisant.

## 6.1 Grand nombre de fautes pour l'algorithme de base

Il existe principalement deux types de problèmes qui peuvent empêcher la convergence de l'algorithme de base en cas de corruption d'un grand nombre de processeurs.

La première est celle dans laquelle la moitié des processeurs sont corrompus et ont la même valeur (exemple (a), figure 6.1). Deux processeurs privilégiés vont alors recevoir des autorisations de mouvement et le démon va pouvoir les faire avancer successivement sans qu'ils se rattrapent. Nous retrouvons là le principal problème de l'exclusion mutuelle, celui des **privilèges surnuméraires**.

A cela vient s'ajouter une autre difficulté, celle de la **configuration terminale**. En effet, dans l'algorithme de base, la procédure de test qui délivre les autorisations de mouvement est une procédure bloquante : elle ne délivre que des réponses positives. En cas de réponse négative, le processeur qui a initié le test ne reçoit rien. Quand un petit nombre de fautes se produit, on a l'assurance qu'un moins un des processeurs n'est pas bloqué. Mais cette certitude disparaît en cas de large corruption. Dans le cas ou chaque processeur privilégié a un corrompu parmi ses $k$ prédécesseurs, tous attendent la réponse d'un test qui ne sera jamais délivrée. Sur l'exemple (b) de la figure 6.1, neuf processeurs privilégiés ont lancé une procédure de test, et les neuf tests sont bloqués.

Pour résoudre ces problèmes, nous présentons un nouvel algorithme. Il peut être considéré comme une amélioration de l'algorithme de base. Seules modifications, le pro-

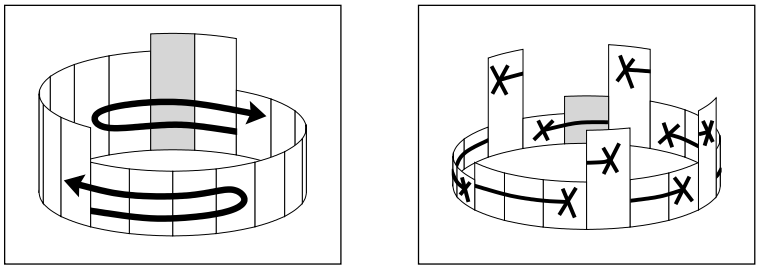



FIG. 6.1 – *L'algorithme de base n'est pas auto-stabilisant*

cesseur distingué retrouve le rôle de filtre à privilèges qu'il avait dans l'algorithme de Dijkstra (pour contrôler les privilèges surnuméraires) et une procédure de détection des blocages est ajoutée.

Ce nouvel algorithme est conçu pour avoir en cas de faible corruption un comportement très proche de l'algorithme de base. Dans tout ce qui va suivre, il convient donc de distinguer deux cas.

1. Le nombre de corruptions est inférieur ou égal à $k$ : l'algorithme de base assure l'anti-corruption et la $k$-convergence. Le nouvel algorithme calque son comportement sur l'algorithme de base. Il utilise pratiquement les même variables et les mêmes règles. Une procédure additionnelle de détection des blocages est exécutée, mais n'a aucune influence (car elle ne détecte aucun blocage).
2. Le nombre de corruptions est strictement supérieur à $k$ : l'algorithme de base n'assure plus l'exclusion mutuelle. En particulier, il ne résout pas les problèmes de privilèges surnuméraires et de blocage. Le nouvel algorithme dispose de mécanismes qui entrent alors en actions et apportent des solutions.
   Il est à noter que dans ce cas, les notions de processeurs corrompus, de faible corruption ou encore de vrais et faux privilèges ne sont plus significatives. Par exemple, parler de vrai et faux privilèges dans la configuration présentée figure 6.1.(a) est sans intérêt puisque les uns comme les autres peuvent recevoir des autorisations de mouvement.

**Blocages :** Pour l'algorithme de base, un blocage (ou configuration terminale) survient lorsqu'un trop grand nombre de processeurs sont privilégiés. Les demandes d'autorisations de mouvement sont lancées, mais aucune n'aboutit.

Plus précisément, lorsqu'il y a blocage, chaque processeur $P$ est dans une des situations suivantes :

1. **S1 :** $P$ est un processeur privilégié et a fait une demande d'autorisation de mouvement. Il attend une réponse.
2. **S2 :** $P$ n'est pas privilégié et a transmis une demande d'autorisation de mouvement. Il attend également une réponse.
3. **S3 :** $P$ n'est pas privilégié et est en attente, $P^+$ lui a transmis une demande d'autorisation de mouvement mais $P^-$ est privilégié et à ce titre, il a engagé sa

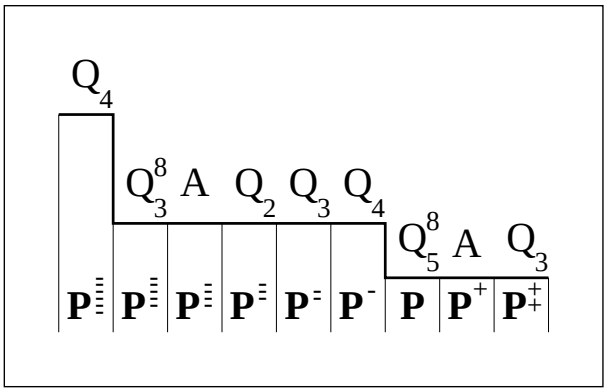



FIG. 6.2 – *Situation de blocage locaux*

propre procédure de test et n'est donc pas prêt à transmettre une question. $P$ attend donc que son prédécesseur soit prêt à transmettre une question.

Dans tous les cas, ces processeurs ne peuvent pas appliquer de règle et tous attendent qu'un autre processeur agisse. Nous dirons d'un processeur qui est dans un des trois cas **S1**, **S2**, **S3** qu'il est localement bloqué. Quand tous les processeurs sont localement bloqués, il y a blocage global, la configuration est terminale.

La figure 6.2 illustre les différents types de blocages locaux possibles pour un processeur. $P$ et $P^{5-}$ sont dans le cas **S1**, $P^{++}$, $P^{-}$, $P^{2-}$, $P^{3-}$ et $P^{6-}$ sont dans le cas **S2**, $P^{+}$ et $P^{4-}$ sont dans le cas **S3**.

En première approche, nous allons résoudre le problème des blocages grâce à un oracle. Quand il y a blocage, les processeurs n'en ont pas connaissance, mais tous sont impliqués dans une procédure de test, et tous sont localement bloqués. Le processeur distingué $D$ ne fait pas exception. Supposons un instant qu'il ait à sa disposition un oracle lui permettant de savoir s'il y a effectivement blocage général. Il peut alors adopter le comportement suivant :

1. S'il n'y a pas blocage général : $D$ ne change rien à son comportement, il suit les règles définies par l'algorithme de base.
2. S'il y a blocage général : $D$ court-circuite la procédure de test dans laquelle il est impliqué et, au lieu de transmettre une demande d'autorisation de mouvement à son prédécesseur, donne une autorisation de mouvement à son successeur.

Ainsi, là où l'algorithme de base aurait été bloqué, le nouvel algorithme assure au plus proche successeur de $D$ possédant un privilège de recevoir une autorisation de mouvement en cas de blocage de tous les autres processeurs.

Techniquement, l'implémentation de l'oracle utilise le fait que les situations de blocage sont localement détectables. Un processeur bloqué est nécessairement dans une des trois situations **S1**, **S2** ou **S3** énumérées ci-dessus. Pour chaque processeur, on peut donc définir le prédicat *BlocageLocal* qui est à vrai quand $P$ est localement bloqué, à faux dans le cas contraire. Ce prédicat est ensuite utilisé par la procédure $TEST$ pour détecter un éventuel blocage global. Plus précisément, si les $n$ prédécesseurs du processeur distingué $D$ sont localement bloqués, c'est que tous les processeurs sont bloqués et

qu'il y a blocage global.

Au final, si après avoir exécuté la procédure $TEST(D,n,BlocageLocal)$, le distingué apprend qu'il y a blocage, il prend la décision de donner une autorisation de mouvement au processeur privilégié qui la lui demande.

Cette procédure assure donc qu'en cas de blocage local généralisé, il existe un privilégié qui reçoit une autorisation de mouvement.

**Privilèges surnuméraires :** Le principe fondamental de l'algorithme de base est de ne pas laisser avancer les faux privilèges. En cas d'un grand nombre de fautes, le test ne détecte plus la nature[1] du privilège qui l'a initié et tous les privilèges peuvent avancer. Si le démon fait alternativement avancer deux privilèges diamétralement opposés sur l'anneau, l'algorithme ne converge plus.

Pour éviter cela, nous utilisons la même idée que celle de l'algorithme de Dijkstra : le processeur distingué $D$ sert de filtre à privilèges.

Dans l'algorithme de Dijkstra, $D$ n'a un privilège que si son prédécesseur $D^-$ a exactement la même valeur que lui-même (ce genre de privilège sera appelé *privilège légitime*). Du point de vue de l'algorithme de base, cette approche n'est pas satisfaisante car un privilège correcteur doit être autorisé à franchir $D$ pour pouvoir rattraper un faux privilège, et cela quelle que soit sa valeur. De même, si un faux privilège est légitime, le distingué ne doit pas le laisser passer, pour permettre à un privilège correcteur de le rattraper.

Le nouvel algorithme mélange les deux techniques : quand $D$ peut détecter la nature d'un privilège, il le laisse passer s'il est correcteur et le bloque s'il est faux. Quand $D$ ne peut pas détecter la nature d'un privilège, il ne laisse passer que les privilèges légitimes.

Concrètement, un distingué ayant un privilège $J$ dispose d'un certain nombre d'éléments lui permettant d'évaluer la situation et de prendre ensuite la décision qui s'impose : transmettre ou ne pas transmettre, telle est la question.

1. $J$ ne reçoit pas d'autorisation de mouvement : $D$ identifie alors clairement son privilège comme un faux privilège et ne le transmet pas.
2. $J$ a une autorisation de mouvement et $D^+$ est dans une situation bloquée : seuls les processeurs engagés dans une procédure de test peuvent être bloqués. Comme $D^+$ est bloqué, $D$ sait qu'il existe un processeur ayant engagé une procédure de test parmi ses $k$ successeurs. $J$ est donc un privilège correcteur et $D$ le transmet.
3. $J$ a une autorisation de mouvement, $D^+$ n'est pas bloqué et $J$ n'est pas un privilège légitime : cela ne peut pas se produire en cas de faible corruption. En effet, si $J$ n'est pas un privilège légitime, c'est que $D$ ou $D^-$ est corrompu et donc que $D$ est a une distance d'au plus $k$ d'un privilège. Or, ce n'est pas le cas, sinon $D^+$ serait bloqué, ou $J$ n'aurait pas reçu d'autorisation de mouvement. $D$ sait donc qu'il est dans une configuration fortement corrompue. Dans ce cas, il se doit de filtrer les privilèges (comme dans l'algorithme de Dijstra). Il ne laisse donc pas passer un privilège non légitime.
4. $J$ a une autorisation de mouvement, $D^+$ n'est pas bloqué et $J$ est un privilège légitime : comme nous venons de le voir, $J$ est loin de tout privilège. Si l'anneau

1. Plus précisément, TEST n'est plus infaillible et peut prendre un faux privilège pour un vrai.

est faiblement corrompu, c'est que $J$ est le vrai privilège et $D$ doit le transmettre. Si l'anneau est fortement corrompu, $D$ doit agir comme le processeur distingué de l'algorithme de Dijstra. Il transmet donc $J$ car $J$ est un privilège légitime.

Au final, $D$ doit donc ne transmettre que les privilèges ayant une autorisation de mouvement. De plus, il doit vérifier que $D^+$ est bloqué ou que le privilège est un privilège légitime. Les différentes situations que le processeur distingué peut rencontrer sont représentées figure 6.3

Ces vérifications permettent l'élimination des privilèges surnuméraires.

## 6.2 Hypothèses & résultats

Nous nous plaçons toujours dans le cadre du **modèle à état** sur un anneau **semi-uniforme bidirectionnel orienté** régi par un démon centralisé.

Sous ses hypothèses, nous présentons un algorithme d'exclusion mutuelle auto-stabilisant. De plus, $k$ étant une constante fixée strictement inférieure à $\sqrt{n}-1$, si le nombre de fautes est inférieur à $k$, il est anti-corruption et converge en $O(kn)$ transitions.

**Théorème 6.2.1**
Soit $k$ une constante strictement inférieure à $\sqrt{n}-1$. L'algorithme d'exclusion mutuelle présenté figure 6.4 est auto-stabilisant. Sa mise en œuvre nécessite $O(log(n))$ bits par processeur. De plus, si le nombre initial de fautes est inférieur à $k$, il est anti-corruption et converge en $O(kn)$ transitions.

## 6.3 Algorithme

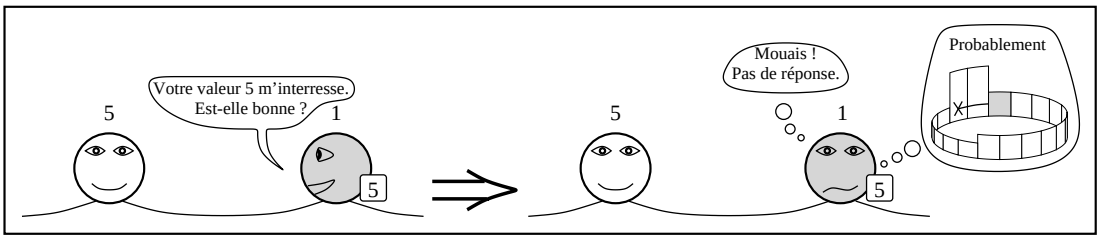
L'algorithme est présenté en deux parties. La figure 6.4.(a) donne l'algorithme de base modifié comme nous venons de le voir pour éviter les blocages. La figure 6.4.(b) présente la procédure de détection des blocages.

**Note :** La procédure $TEST(P,i,Predicat)$, définie dans la section précédente, permet a un processeur de vérifier que ses $i$ prédécesseurs satisfont $Predicat$. Ces principales caractéristiques sont :

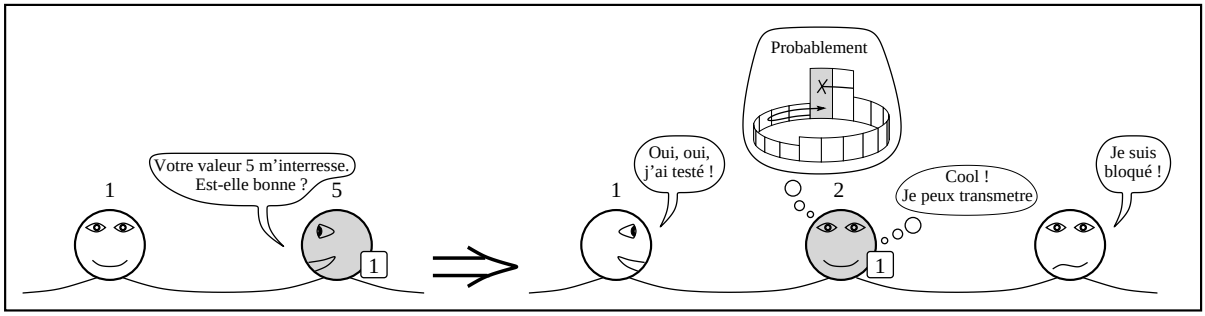
1. $TEST$ assure à un processeur non initialement corrompu une réponse correcte.
2. Implémenter $TEST$ nécessite une variable de taille $O(i)$.
3. La terminaison de $TEST$ est assurée en $O(i)$ transitions.

Il est à noter que la propriété détectée par le test, par exemple la non présence de privilège, n'intervient pas dans le fonctionnement de la procédure : seuls participent au test les processeurs vérifiant le $Predicat$. Si le test aboutit, c'est que tous les processeurs testés le satisfont. Sinon, le test est bloqué.

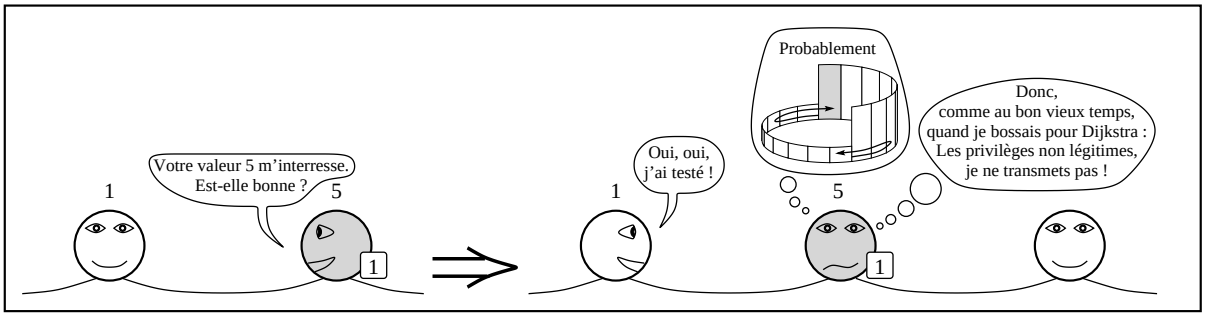
Dans l'algorithme de base, $TEST(P,k,NonPrivilege)$ est utilisé par un processeur $P$ pour détecter d'éventuels privilèges parmi ses $k$ prédécesseurs. Dans cette nouvelle version de l'algorithme, nous passons sous silence les détails du fonctionnement de la



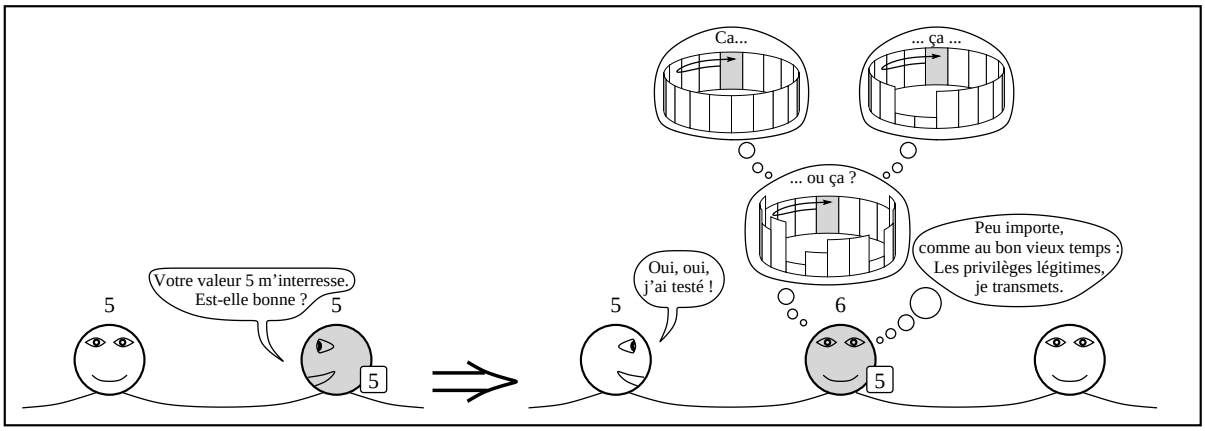

(a) Pas d'autorisation de mouvement




(b) Autorisation de mouvement ET successeur bloqué




(c) Autorisation de mouvement, successeur non bloqué et privilège non légitime




(d) Autorisation de mouvement, successeur non bloqué et privilège légitime

FIG. 6.3 – *L'art d'être un bon processeur distingué*

**Prédicat : Définition d'un Privilège**

$Privilege(P) \Leftrightarrow \begin{cases} Priv(P) \neq Priv(P^-) + 1 & \text{si } P \text{ est le processeur distingué.} \\ Priv(P) \neq Priv(P^-) & \text{si } P \text{ est un processeur quelconque.} \end{cases}$

$PrivLegitime(D) \Leftrightarrow Priv(D) = Priv(D^-)$ (défini uniquement pour le distingué)

**Prédicats pour un processeur quelconque**

$BlocageLocal(P) \Leftrightarrow \begin{cases} Test(P) = (Q_k^j) & \text{si } P \text{ est privilégié} \\ Test(P^-,P) \in \{(A,Q_i),(Q_{i-1},Q_i),(Q_k^j,A)\} & \text{sinon (avec } i \geq 2) \end{cases}$

**Prédicats pour un privilégié**

$Autorisation(P) \Leftrightarrow TEST(NonPrivilege,P,k)$ est vrai

**Règles gardées pour un processeur non distingué**

$R1$ $Privilege(P)$ et $Autorisation(P) \Rightarrow Priv(P) \longleftarrow Priv(P^-)$

**Règles gardées pour le processeur distingué**

$R1$ $Privilege(P)$, $Autorisation(P)$ et $BlocageLocal(P^+) \Rightarrow Priv(P) \longleftarrow Priv(P^-) + 1$

$R2$ $Privilege(P)$, $Autorisation(P)$ et $PrivLegitime(P) \Rightarrow Priv(P) \longleftarrow Priv(P^-) + 1$

FIG. 6.4 – *Exclusion mutuelle auto-stabilisante et k-stabilisante*

**Prédicats pour le distingué**

$Blocage(D) \Leftrightarrow TEST(Blocage,D,n)$

**Action pour le distingué**

$Deblocage(P) \Leftrightarrow \begin{cases} Test(D) \longleftarrow R & \text{Si } D \text{ n'a pas de privilège} \\ Priv(D) \longleftarrow Priv(D^-) + 1 & \text{Si } D \text{ a un privilège} \end{cases}$

**Règles gardées**

$RB$ $Blocage(D) \Rightarrow Deblocage(P)$

FIG. 6.5 – *Procedure de detection des blocages*

procédure. Au lieu de manipuler les variables $Test(P^-,P,P^+)$ et les nombreuses règles qui s'y réfèrent, nous utilisons directement le prédicat $TEST(P,k,NonPrivilege)$, prédicat qui est vrai quand aucun des $k$ prédécesseurs de $P$ n'a de privilège. De la même manière, le prédicat $TEST(P,n,BlocageLocal)$ est vrai quand les $n$ prédécesseurs de $P$ ont tous leur variable $BlocageLocal$ à vrai.

Cette facilité d'écriture s'étend aux exécutions : seules les transitions concernant les règles $R1$, $R2$ et $RB$ sont considérées. Ces transitions sont appelées *mouvements de privilèges*. Néanmoins, il est important de ne pas perdre de vue la réalité des choses quand au coût du test : l'implémentation de $TEST(P,n,BlocageLocal)$ nécessite une variable de taille $O(n)$ et la convergence de la procédure (assurée par une fonction de potentiel du test, définition 5.2.4, page 89) se fait en $O(n)$ transitions. Ainsi, chaque mouvement autorisé par un processeur quelconque cache un surcoût de $O(k)$ transitions (exécution de $TEST(P,n,NonPrivilege)$) et les autorisations de mouvement du distingué nécessitent potentiellement $O(n)$ transitions (exécution de $TEST(P,n,BlocageLocal)$).

**Variables :** Chaque processeur dispose de trois variables. Comme dans l'algorithme de base, $Priv(P)$, à valeur dans $[0..n]$, détermine l'éventuelle presence d'un privilége alors que $Test(P)$, à valeur dans $\{A\} \cup \{R\} \cup \{Q_i\}_{i\in[1..k-1]} \cup \{Q_k^i\}_{i\in[0..k]}$, permet le fonctionnement de $TEST(P,k,NonPrivilege)$. La troisième variable est utilisée pour l'implémentation de $TEST(D,k,BlocageLocal)$. Elle est à valeur dans $\{A\} \cup \{R\} \cup \{Q_i\}_{i\in[1..n]}$.

**Le nouveau Protocole :** La nouvelle version du protocole de base est présenté sous forme de prédicats permettant de définir des règles gardées.

1. **Privilege**(**P**) est vrai quand $P$ a un privilège. $D$ a un privilège quand sa variable $Priv(D)$ est différente de $Priv(D^-)+1$ alors que les autres processeurs sont privilégiés quand leur variable $Priv(P)$ est différente de celle de leur prédécesseur.
2. **PrivLegitime**(**D**) n'est défini que pour le processeur distingué. Ce prédicat est vrai quand $J$, le privilége de $D$, est un privilège légitime, c'est à dire quand la valeur du privilège $J$ est la même que celle de $D$.
3. **BlocageLocal**(**P**) est vrai quand $P$ est impliqué dans une procédure de detection des privilèges et qu'il est en attente (d'une réponse ou de la possibilité de transmettre une question).
4. **Autorisation**(**P**) est vrai quand $P$ a fait appel à la procédure $TEST(P,k,NonPrivilege)$ et que celle-ci n'a pas détecté de privilège parmi les $k$ prédécesseurs de $P$. Elle lui délivre alors une autorisation de transmission.

La régle de transmission de privilège est différente pour un processeur quelconque et pour le distingué :

1. Pour un processeur non distingué, **R1** : cette règle est la même que la règle $R1$ de l'algorithme de base : un processeur qui a un privilège et l'autorisation de transmettre transmet.
2. Pour le processeur distingué, **R1** : cette règle autorise le processeur distingué à transmettre son privilège quand son successeur est impliqué dans une procédure de test.
3. Pour le processeur distingué, **R2** : cette règle autorise le processeur distingué à transmettre les privilèges légitimes.

**Procédure de déblocage :** Grâce à l'utilisation de la procédure $TEST$, la description de la procédure de déblocage est assez simple :

1. **Blocage**(**D**) est vrai quand le processeur distingué, après avoir déclenché un test pour détecter un blocage total, reçoit une réponse positive à sa requête.
2. L'action **Deblocage**(**D**) consiste pour le processeur distingué à transmettre son privilège s'il en a un, a donner une autorisation de transmission à son successeur sinon.
3. La règle **RB** consiste pour le processeur distingué à autoriser une transmission de privilège (le sien ou celui d'un de ses successeurs) en cas de detection d'un blocage global.

## 6.4 Preuve

### 6.4.1 Configurations légitimes

**Définition 6.4.1**
L'ensemble des configurations légitimes est l'ensemble des configurations dans lesquelles un seul processeur est privilégié.

### 6.4.2 Correction

La preuve de la correction est exactement la même que celle de l'algorithme de base : le nombre de privilège n'augmente jamais, il existe toujours au moins un privilège sur l'anneau, il n'existe pas de configuration terminale n'ayant qu'un seul privilège. Au final, l'ensemble des configurations légitimes est inclus dans l'ensemble des configurations correctes.

### 6.4.3 Complexité de la $k$-convergence

L'étude de la complexité de la $k$-convergence utilise, comme pour l'algorithme de base, une fonction de potentiel de test. Unique modification, les autorisations de transmissions délivrées par $D$ nécessitent l'exécution de la procédure de détection des blocages locaux. Mais comme seulement $k$ processeurs sont corrompus, cette procédure s'exécute en $O(k)$ transitions, c'est à dire dans un temps identique à celui de l'exécution de la procédure détectant les priviliéges. Cela ne change donc pas la complexité globale.

### 6.4.4 Convergence

**Définition 6.4.2**
Un privilège est *en panne* si un de ses $k$ prédécesseurs possède également un privilège. Un privilège non en panne est *libre*.

**Définition 6.4.3**
On appelle configuration *bloquée* une configuration dans laquelle tous les privilèges sont en panne.

**Lemme 6.4.4**
Si dans une configuration tous les processeurs sont localement bloqués, alors la configuration est bloquée.
Réciproquement, si une exécution $\mathcal{E}$ a pour configuration initiale une configuration bloquée, alors il existe dans $\mathcal{E}$ une configuration dans laquelle tous les processeurs sont localement bloqués.

**Preuve :** Soit $C$ une configuration et $P$ un privilégié. $P$ est localement bloqué, d'ou $Test(P) = Q_k^j$. Les $k$ predecesseurs de $P$ sont tous localement bloqués. Soit $P^{i-}$ le premier prédécesseur de $P$ dont la variable $Test$ n'est pas sur $Q$. $P^{i-}$ est nécessairement parmi les $k-1$ prédécesseurs de $P$, car les indices des variables $Test$ sont décroissants de $k$ à 2. De plus, on a $Test(P^{(i+1)-},\mathbf{P^{i-}},P^{(i-1)-} = (Q_k^j,\mathbf{A},Q)$ car les trois processeurs $P^{(i+1)-}$,$P^{i-}$ et $P^{(i-1)-}$ sont tous localement bloqués. Or, seul un processeur privilégié peut être bloqué en ayant sa variable à $Q_k^j$. Au final, $P^i$ faisant partie des $k-1$ prédécesseurs de $P$, $P^{(i+1)-}$ est parmi les $k$ prédécésseurs de $P$. □

**Réciproquement :** Si un processeur privilégié en panne initie un test, alors le test n'aboutit pas et tous les processeurs impliqués dans le test deviennent localement bloqués.

Soit $P$ un processeur et soit $P'$ son plus proche successeur à être privilégié. $P$ est parmi les $k$ prédécesseurs de $P'$ (sinon, $P'$ serait un processeur privilégié n'ayant pas de privilège parmi ses $k$ prédécesseurs, ce qui est impossible car $P$ est en panne). D'où $P$ participe au test de $P'$ et, après un certain nombre de transitions, $P$ sera localement bloqué. □

**Lemme 6.4.5**
Soit $\mathcal{E}$ une exécution dont la configuration initiale est bloquée. Soit $P$ le plus proche successeur de $D$ (éventuellement $D$ lui-même) possédant un privilège. Alors il existe dans $\mathcal{E}$ une transition impliquant le mouvement du privilège de $P$.

**Preuve :** Il existe dans $\mathcal{E}$ une configuration dans laquelle tous les processeurs sont localement bloqués (lemme 6.4.4). Dans cette configuration, $TEST(D,n,Blocage)$ est vrai. $D$ va donc appliquer la règle $RB$ suite à quoi $D$ transmettra son privilège s'il en a un, donnera à $P$ une autorisation de mouvement sinon. □

**Lemme 6.4.6**
Soit $\mathcal{E}$ une exécution dont la configuration initiale est bloquée. Alors $\mathcal{E}$ contient une configuration dont un privilège est libre.

**Preuve :** Il existe dans $\mathcal{E}$ une transition impliquant le mouvement du plus proche privilège $J$ suivant $D$ (lemme 6.4.6). Après cette transition, si $J$ est toujours en panne, le même lemme assure qu'une autre transition permet à $J$ un autre mouvement. Après au plus $k$ transitions de ce genre, $J$ est à distance $k$ de $D$. Comme tous les autres privilèges sont en panne, aucun n'a pu franchir $D$. $J$ est donc libre. □

**Lemme 6.4.7**
Soit $T = C_1 \xrightarrow{(P,R)} C_2$ un mouvement de privilège. S'il existe un privililège libre dans $C_1$, alors il existe également un privilège libre dans $C_2$.

**Preuve :** En fonction de l'éventuelle présence d'un privilège en $P^+$, deux cas sont possibles :

1. $P^+$ a reçu le privilège $J$ de $P$ et a maintenant un privilège : dans ce cas, $J$ était libre dans $C_1$, il l'est également dans $C_2$ puisqu'aucun de ses $k+1$ prédécesseurs n'a de privilège.
2. $P$ a transmis son privilège à $P^+$, mais celui-ci avait déjà un privilège et les deux privilèges ont disparu en fusionnant : soit $P'$ le plus proche successeur de $P^+$ ayant un privilège. Ce privilège est libre puisque parmi des processeurs compris entre lui et $P$ *et* les $k$ prédécesseurs de $P$, on ne trouve aucun privilégié.

Dans tous les cas, un privilège libre existe dans $C_2$. □

**Corollaire 6.4.8**
Soit $\mathcal{E}$ une exécution et $C_0$ sa configuration initiale. S'il existe un privilège libre dans $C_0$, alors il existe un privilège libre dans chacune des configurations de $\mathcal{E}$.

**Lemme 6.4.9**
Soit une exécution dont la configuration initiale n'est pas bloquée. Alors au cours de l'exécution, un privilège en panne ne peut devenir libre qu'après la disparition de deux privilèges.

**Preuve :** Un privilège $J$ en panne ne peut pas recevoir d'autorisation de mouvement (pas même le plus proche privilégié suivant $D$, la configuration initiale n'est pas bloquée, $TEST(D,n,BlocageLocal)$ est faux en permanance), il ne peut donc pas bouger. De plus, la circulation des privilèges ne se faisant que dans un sens, le privilège $J'$ présent parmi les $k$ prédécesseurs de $J$ ne peut que se rapprocher de $J$. Un privilège ne peut disparaitre que s'il fusionne avec un autre et que la fusion entraîne la disparition des deux privilèges. Si $J$ devient libre, c'est donc que deux privilèges ont fusionné et disparu. □

**Lemme 6.4.10**
Soit $\mathcal{E}$ une exécution. Soit $\mathcal{E}'$ une portion de $\mathcal{E}$ comportant au moins $2n^2$ mouvements de privilège. Alors il existe dans $\mathcal{E}'$ une configuration ayant au moins un privilège en panne.

**Preuve :** Par l'absurde : supposons que dans toutes les configurations de $\mathcal{E}'$, tous les privilèges soient libres. Le processeur distingué ne peut donc jamais exécuter la règle $R1$ (car $R1$ ne sert que quand le privilège suivant le distingué est en panne). Donc, si $D$ applique une règle de transmition de privilège, c'est la règle $R2$, c'est à dire la règle laissant passer les privilèges légitimes. L'algorithme a donc le même comportement que celui de Dijstra.

Comme l'exécution $\mathcal{E}'$ ne contient aucun privilège en panne, elle ne peut pas contenir de fusion, pas plus que de configuration légitime. Soit $J$ un privilège. Si $J$ franchit $D$ deux fois, l'exécution atteint une configuration légitime (lemme 3.5.9, page 65). Donc dans $\mathcal{E}'$, $J$ peut au plus avancer $2n-1$ fois. Il en est de même pour tous les privilèges de $\mathcal{E}'$. Or, une configuration comportant moins de $n$ privilèges, $\mathcal{E}'$ peut donc contenir au plus $n \times (2n-1)$ mouvements de privilèges. C'est contraire à l'hypothèse de départ. $\mathcal{E}'$ contient donc au moins une configuration dont un processeur est en panne.

□

**Lemme 6.4.11**
Soit $C$ une configuration dans laquelle $i$ privilèges sont présents. Si un des privilèges est en panne, après au plus $in$ mouvement de privilèges, deux privilèges fusionnent.

**Preuve :** Un privilège $J$ en panne ne peut bouger. Donc un autre processeur qui avance ne peut faire plus de $n$ mouvements (un tour complet d'anneau) sans fusionner avec $J$. Or, si $i$ privilèges totalisent $in$ mouvements, c'est que l'un d'entre eux au moins en fait $n$ ou plus. Ce privilège fusionne nécessairement avec un autre.

□

**Lemme 6.4.12**
Soit $\mathcal{E}$ une exécution. En moins de $2n^3$ mouvements de privilèges, $\mathcal{E}$ atteint une configuration légitime.

**Preuve :** Soit $C_0$ la configuration initiale de l'exécution et $i$ le nombre initial de privilèges.

Après au plus $2n^2$ mouvements de privilèges, un privilège est en panne (lemme 6.4.10). Après $i \times n$ mouvement de privilèges, deux privilèges ont fusionné (lemme 6.4.11). L'exécution est alors dans la configuration $C_1$ dans laquelle au plus $i-1$ privilèges sont présents.

En itérant le processus, on construit $C_{i-1}$, une configuration dans laquelle $i-(i-1)$ privilèges sont présents, c'est à dire une configuration légitime.

$C_{i-1}$ est atteinte en moins de

$$\begin{aligned} \textstyle\sum_{j=i}^{1}\{n^2+jn\} &= in^2 + n\textstyle\sum_{j=1}^{i} j \\ &= in^2 + n(i(i-1)/2) \\ &< n^3 + n^3 \end{aligned}$$

mouvements de privilèges.

□

**Lemme 6.4.13**
L'algorithme converge en $O(kn^3)$ transitions.

**Preuve :** Après $n^3$ mouvements de privilège, l'algorithme a convergé. Chacun de ses mouvements a nécessité l'exécution de la procédure de detection des privilège (coût $O(k)$ transition).

D'autre part, parmi ces $n^3$ mouvements, au plus $n^2$ ont concerné le privilège distingué et donc $kn^2$ d'entre eux ont potentiellement déclenché l'exécution de la procédure de detection des blocages (coût $O(n)$ transition).

Au final, la convergence est assurée après $O(k \times n^3 + n \times kn^2)$ transissions. □

### 6.4.5 Bilan

**Théorème 6.4.1**
Soit $k$ une constante strictement inférieure à $\sqrt{n} - 1$. L'algorithme d'exclusion mutuelle présenté figure 6.4, page 111 est auto-stabilisant. Sa mise en œuvre nécessite $O(log(n))$ bits par processeur. De plus, si le nombre initial de fautes est inférieur à $k$, il est anti-corruption et converge en $O(kn)$ transitions.

**Preuve :** Quand le nombre de fautes est quelconque, la convergence est assurée (lemme 6.4.13). Quand le nombre de fautes est inférieur à $k$, l'algorithme est identique à l'algorithme de base, ce qui assure sa correction, sa convergence rapide et son anti-corruption. □

# Deuxième partie

# Le Cas Synchrone

*ou*

# Course-poursuite sur un anneau

# Chapitre 7

# Exclusion mutuelle auto-stabilisante synchrone

Cette partie présente des techniques permettant d'améliorer les performances des algorithmes auto-stabilisants fonctionnant avec un démon synchrone. Encore une fois, notre étude se base sur le problème de l'exclusion mutuelle.

Par rapport aux algorithmes centralisés ou répartis, le cas synchrone présente deux avantages. En premier lieu, le démon est déterministe : à chaque transition, tous les processeurs activables agissent. Dans le modèle à états, les délais de communication ne sont pas pris en compte. Il en résulte qu'une configuration ne peut conduire qu'à une unique configuration. Généralisé aux exécutions, cette propriété assure le déterminisme de l'exécution[1].

Autre avantage, le synchronisme permet d'ajouter une importante dimension aux algorithmes, celle du temps. En effet, les processeurs sont munis d'une horloge. Dès lors, des notions de vitesse absolue ou de temps de propagation d'information peuvent être définies. Notamment, certains processeurs peuvent volontairement agir moins fréquemment que d'autres pour laisser à des informations le temps de leur parvenir.

C'est ce genre de technique qu'utilise l'algorithme de ce chapitre. Tout comme lors du chapitre 3 dans le cas asynchrone, il présente les concepts de base qui seront ensuite utilisés par les algorithmes synchrones plus complexes des chapitres suivants.

## 7.1 Idées générales

L'algorithme que nous présentons ici résout le problème de l'exclusion mutuelle sur un anneau. Cette fois ci, le démon considéré est synchrone. Dans le cas synchrone, les problèmes à résoudre sont les mêmes que ceux du cas centralisé, à savoir assurer l'existence d'un privilège et la disparition des privilèges surnuméraires.

Pour **l'existence d'un privilège**, la technique utilisée est la même que celle de Dijkstra : de par la définition même du privilège, il en existe un dans toutes les configurations.

1. Nous ne considérons ici que les algorithmes dont les régles gardées sont déterministes.

Pour **l'élimination des privilèges surnuméraires**, nous proposons une approche différente de celle du précédent chapitre. Elle se base sur l' interprétation dynamique de l'existence et de la disparition des privilèges (section 2.4.3, page 52). Sous certaines conditions, on peut assurer que 1) le nombre de privilèges présents sur un anneau est toujours impair et que 2) lorsqu'un privilège arrive sur un processeur possédant déjà un privilège, les deux privilèges disparaissent.

Résoudre le problème des privilèges surnuméraires revient alors à assurer que certains privilèges en rattrapent d'autres. D'où l'idée de donner aux divers privilèges des vitesses de progression différentes. Cela est rendu possible par la caractéristique synchrone du démon. En effet, un démon synchrone se doit de faire agir tous les processeurs simultanément. D'où, un privilège qui serait transmis à chaque round et un privilège qui ne serait transmis que tous les deux rounds auraient des vitesses différentes. C'est ce genre de mécanisme que met en jeu notre algorithme d'exclusion mutuelle synchrone.

## 7.2 Hypothèses & résultats

Les hypothèses de cet algorithme sont les mêmes que pour celui de Dijkstra : toujours dans le **modèle à état**, le graphe considéré est un **anneau semi uniforme unidirectionnel orienté**. Seule change la spécification du démon : nous considérons ici un **démon synchrone** (à chaque round, tous les processeurs exécutent leur règle).

Sous ces hypothèses, nous proposons un algorithme d'exclusion mutuelle auto-stabilisant ayant un temps de stabilisation proportionnel à la taille de l'anneau et optimal en espace mémoire :

**Théorème 7.2.1**
L'algorithme présenté figure 7.1 est auto-stabilisant pour le problème de l'exclusion mutuelle. De plus, si la taille de l'anneau est $n$, sa mise en application nécessite 1 bit par processeur[2] et son temps de stabilisation est de $O(n)$.

## 7.3 Algorithme

L'algorithme est présenté figure 7.1.

Chaque processeur $P$ dispose d'une variable $Priv(P)$ déterminant la position du privilège. $Priv(P)$ est à valeur dans [0..1]. Les opérations d'incrémentations de $Priv(P)$ se font modulo 2.

**Prédicats :** on dit du processeur distingué $D$ qu'il a le privilège quand sa variable $Priv(D)$ a la même valeur que celle de son prédécesseur. Un processeur non-distingué $P$ a

2. nous considérons que chaque processeur est doté d'une horloge interne propre lui permettant de différer de un round l'exécution d'une action. Sans cette hypothèse, la mise en place explicite d'une horloge se fait en octroyant à chaque processeur une variable "compteur" à valeur dans [0..1]. La complexité en espace est alors de 2 bits par processeur.

**Prédicats :**

$Privilege(P) \quad \Leftrightarrow \begin{cases} Priv(P^-) = Priv(P) & \text{si } P \text{ est le processeur distingué.} \\ Priv(P^-) \neq Priv(P) & \text{si } P \text{ est un processeur quelconque.} \end{cases}$

$PrivilegeRapide(P) \quad \Leftrightarrow Privilege(P)$ et $Priv(P) = 0$

$PrivilegeLent(P) \quad \Leftrightarrow Privilege(P)$ et $Priv(P) = 1$

**Action :**

$TransmetPrivilege(P) \quad \Leftrightarrow Priv(P) \longleftarrow Priv(P) + 1 \ \ (modulo 2)$

**Règles gardées :**

Tous les rounds : $PrivilegeRapide(P) \implies TransmetPrivilege(P)$

Tous les 2 rounds : $PrivilegeLent(P) \implies TransmetPrivilege(P)$

FIG. 7.1 – *Exclusion mutuelle synchrone*

le privilège quand sa variable $Priv(P)$ a une valeur différente de celle de son prédécesseur $Priv(P^-)$.

**Nature des privilèges :** un privilège est rapide si le processeur le possédant a sa variable $Priv$ à 0. Sinon, il est lent. Un privilège rapide est transmis de processeur en processeur tous les rounds. Un privilège lent est transmis seulement tous les deux rounds : lorsqu'il vient d'arriver sur un processeur $P$, il est "lent fatigué". Lors de la transition suivante, il devient "lent reposé". Lors de la transition suivante, il est transmis. Il arrive alors sur $P^+$ où il est à nouveau "lent fatigué". Ainsi de suite.

**Action :** pour seule action, un processeur peut modifier la valeur de sa variable $Priv(P)$.

**Règles :** les règles sont simples : elles font avancer les privilèges rapides tous les rounds, et les privilèges lents tous les deux rounds.

L'algorithme est auto-stabilisant. Son bon fonctionnement est lié à un certain nombre de propriétés (illustrées figure 7.2).

**Propriétés 7.3.1**

1. La variable $Priv(P)$ qui détermine l'existence de privilèges ne pouvant prendre que deux valeurs, le nombre de privilèges est toujours impair (voir figure 7.2)
2. Dans une configuration donnée, si l'on parcourt (visuellement) l'anneau dans un sens quelconque en partant de $D$, on ne rencontre jamais successivement deux privilèges de même nature (rapide ou lent) (voir figure 7.2, configurations 2, 3 et 4).
3. Lorsqu'un privilège arrive en $D$, sa nature (rapide ou lent) change.

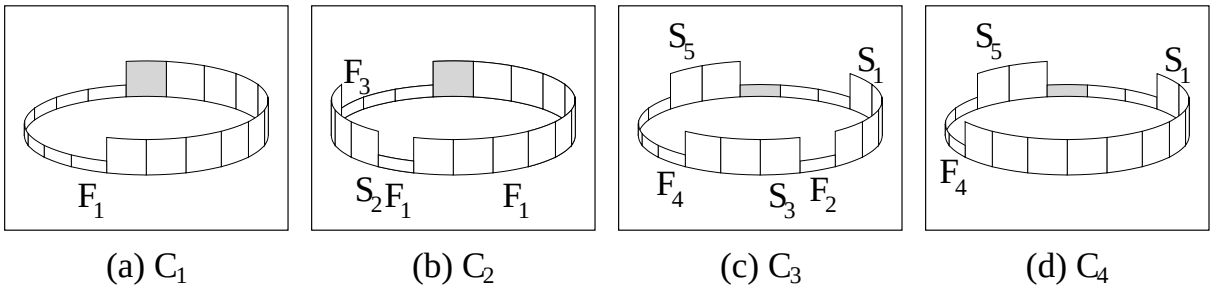



FIG. 7.2 – *Exemple de configurations*

4. Lorsque deux privilèges fusionnent, ils disparaissent tous les deux. Par exemple, dans la transition $C_3 \longrightarrow C_4$ de la figure 7.2, le privilège $F_2$ est transmis alors que $S_3$ reste immobile. Au final, dans la configuration 4, les deux privilèges ont disparu.

### 7.3.1 Exemples d'exécutions

La figure 7.3 présente deux exécutions possibles du système. La première est une exécution légitime. Un seul processeur est privilégié. Quand le privilège est rapide ($F_1$), il est transmis tous les rounds. Quand il est lent ($S_1$) il est transmis tous les deux rounds.

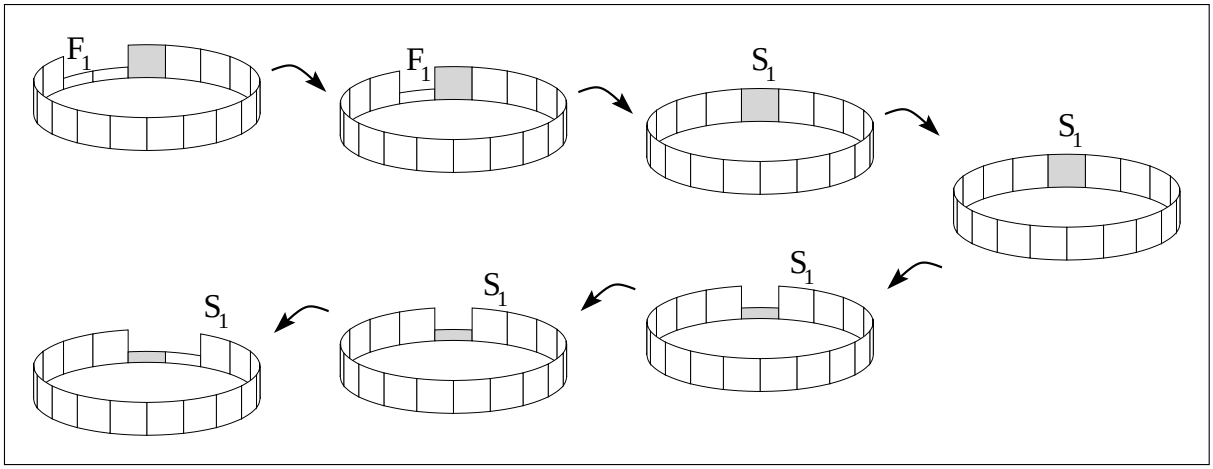
La deuxième exécution a pour configuration initiale une configuration corrompue. Initialement cinq privilèges sont présents. Mais à chaque fois qu'un privilège rapide fusionne avec un lent, les deux privilèges disparaissent.

## 7.4 Preuve

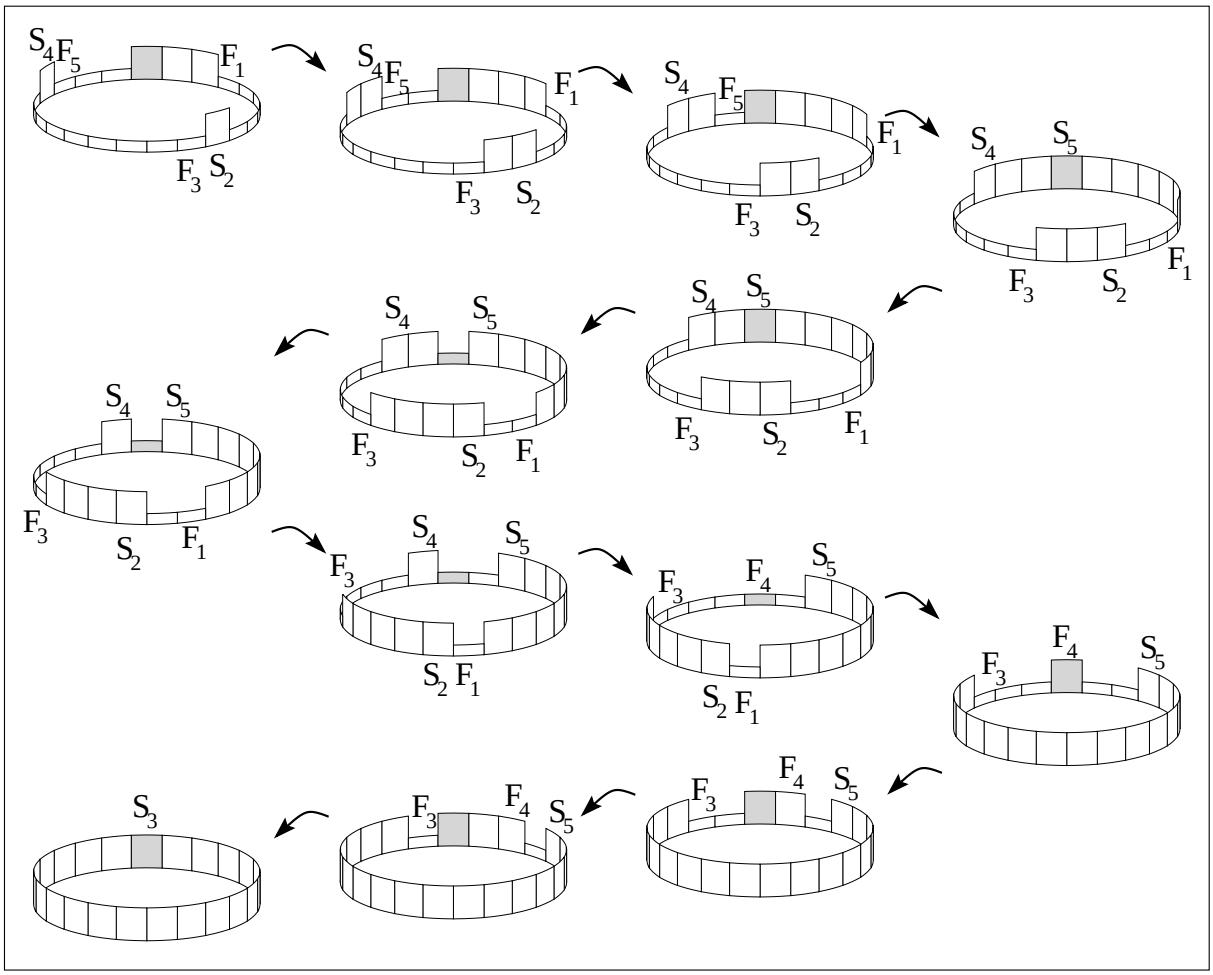
**Notations et conventions :** les notations présentées ici sont illustrées figure 7.3.

Soit $C$ une configuration. Dans tout ce qui suit, $J$ représente un privilège (rapide ou lent), $F$ représente un privilège rapide[3] alors que $S$ représente un privilège lent[4]. Si plusieurs privilèges sont présents dans $C$, nous les numérotons en partant de $D$ et en parcourant l'anneau dans le sens indirect. Selon les cas, certains privilèges peuvent être omis de l'énumération (par exemple, si nous parlons de deux privilèges $F_1$ et $S_2$, il peut exister des privilèges entre $F_1$ et $S_2$). Dans le cadre d'une exécution, les privilèges numérotés dans une configuration conservent le même numéro tout au long de l'exécution. Notamment, un privilège qui franchit $D$ change de nature mais ne change pas de numéro (par exemple, si $F_3$ franchit $D$, il devient $S_3$).

3. $F$ pour "Fast", la lettre $R$ étant déjà utilisée pour désigner les règles de l'algorithme.
4. $S$ pour "Slow", la lettre $L$ étant déjà utilisée pour désigner les configurations légitimes.




(a) Exécution légitime




(b) Exécution non légitime

FIG. 7.3 – *Exemples d'exécution*

### 7.4.1 Configuration légitime

L'ensemble des configurations légitimes choisi est l'ensemble des configurations dans lesquelles un seul processeur a le privilège.

**Définition 7.4.1**
Une configuration $L$ est légitime si et seulement si il existe un unique processeur dans $C$ possédant un privilège.

### 7.4.2 Correction

La preuve de la correction est la même que celle de l'algorithme de Dijkstra.

**Lemme 7.4.2**
Dans toute configuration, il existe au moins un processeur privilégié.

**Preuve :** Même preuve que celle du lemme 3.5.2 page 63. □

**Lemme 7.4.3**
Au cours d'une exécution, le nombre de privilèges n'augmente jamais.

**Preuve :** Même preuve que celle du lemme 3.5.4 page 63. □

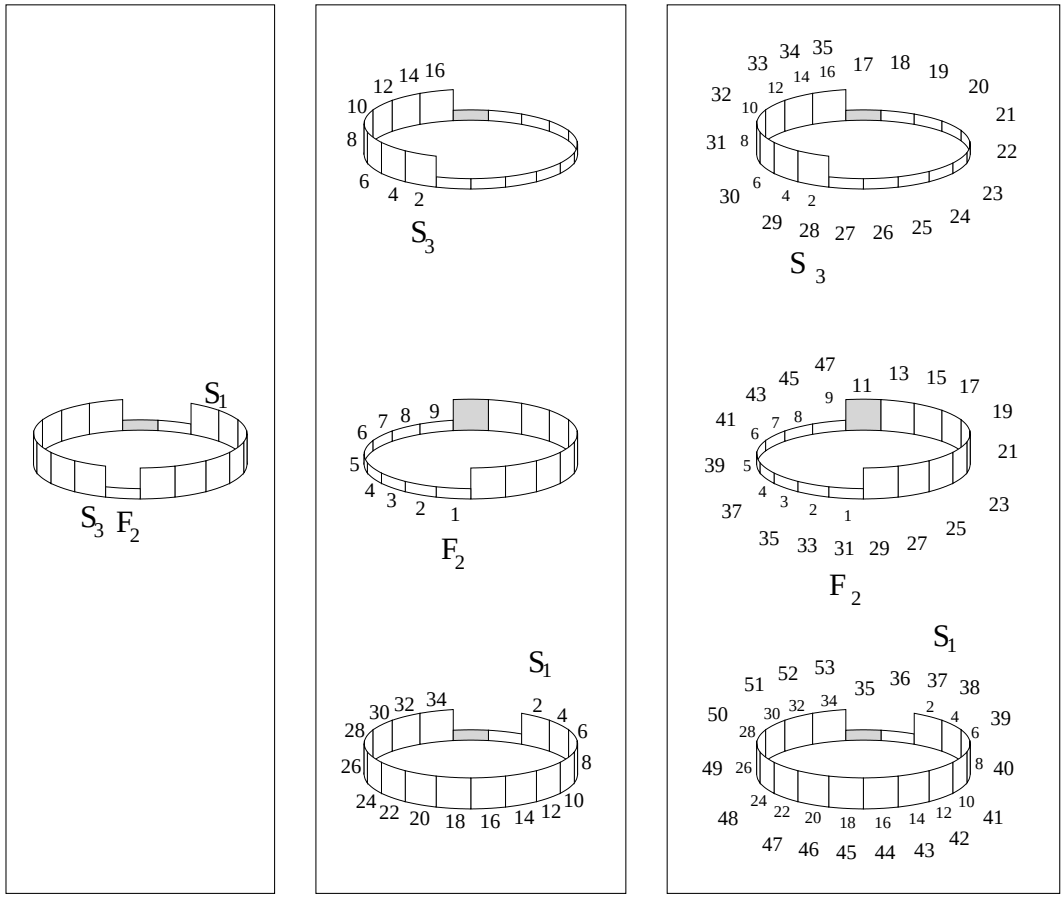
**Lemme 7.4.4**
Toute exécution dont la configuration initiale est légitime vérifie la spécification de l'exclusion mutuelle.

**Preuve :** Le nombre de privilèges ne pouvant être nul (lemme 7.4.2) et n'augmentant pas (lemme 7.4.3), il reste constant et égal à 1 durant toute l'exécution. □

### 7.4.3 Convergence

Soit $P$ un processeur et $J$ son privilège. Si $J$ n'interfère avec aucun autre privilège, alors après un certain nombre de transitions, il franchit le processeur distingué $D$. Informellement, $RejointD(J)$ est le nombre de rounds nécessaires à $J$ pour atteindre $D$ s'il n'est pas perturbé par un autre privilège dans sa marche.

De même, $RejointDD(J)$ est le nombre de rounds nécessaires à $J$ pour atteindre le processeur distingué $D$ une première fois, puis faire un tour complet de l'anneau et atteindre $D$ une seconde fois.




(a) Configuration $C_0$ (b) RejointD (c) RejointDD

FIG. 7.4 – *RejointD et RejointDD*

**Définition 7.4.5**
Soit $C$ une configuration et $J$ un privilège de $C$. $RejointD(J)$ est la distance séparant $J$ de $D$ si $J$ est rapide, 2 fois cette distance si $J$ est lent.

$$RejointD(J) = \begin{cases} Dist(J,D) & \text{si } J \text{ est un privilège rapide.} \\ 2 \times Dist(J,D) & \text{si } J \text{ est un privilège lent au repos.} \\ 2 \times Dist(J,D) - 1 & \text{si } J \text{ est un privilège lent actif.} \end{cases}$$

$$RejointDD(J) = \begin{cases} RejointD(J) + 2n & \text{si } J \text{ est un privilège rapide.} \\ RejointD(J) + n & \text{si } J \text{ est un privilège lent.} \end{cases}$$

Un exemple est présenté figure 7.4. Dans l'exécution dont la configuration initiale est $C_0$, le privilège $S_3$ ne parvient jamais à $D$, il est rejoint par $F_2$ avant cela. Néanmoins, sa variable $Rejoint(S_3)$ vaut 16 car, s'il ne fusionnait pas avec un autre processeur, $S_3$ mettrait 16 rounds pour parvenir à $D$.

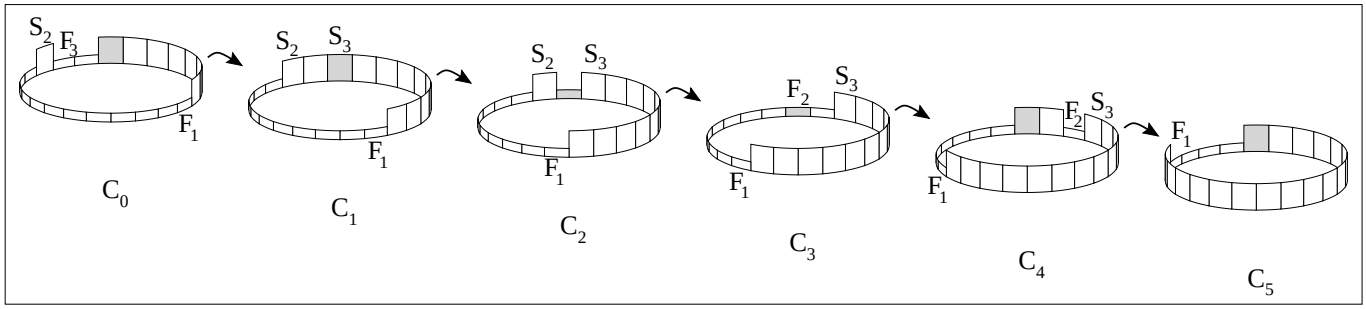



FIG. 7.5 – *Illustration des lemmes 7.4.6, 7.4.7 et 7.4.8*

**Lemme 7.4.6**
Soit $\mathcal{E}$ une exécution et $C$ une configuration de $\mathcal{E}$. Soit $F_1$ un privilège rapide et $S_2$ un privilège lent de $C$. Si $RejointD(F_1) \leq RejointD(S_2)$, alors $S_2$ disparaîtra avant de franchir $D$.

**Preuve :** Supposons qu'il n'y ait pas de privilèges entre $F_1$ et $S_2$ (dans le cas contraire, tout ce qui suit s'appliquerait à $F_1'$ et $S_2$, avec $F_1'$ le privilège rapide situé entre $F_1$ et $S_2$ le plus proche de $S_2$). $F_1$ est un privilège rapide. Il ne peut donc pas disparaître en étant rattrapé avant d'avoir franchi $D$. D'autre part, le fait que $RejointD(F_1)$ soit plus petit que $RejointD(S_2)$ signifie que $F_1$ devrait arriver à $D$ avant $S_2$. Comme deux privilèges ne peuvent pas se doubler sur l'anneau, $F_1$ est sûr de rattraper $S_2$ avant que celui-ci ne parvienne en $D$. □

Une illustration de ce lemme est donné figure 7.5. Dans la configuration $C_3$, $RejointD(F_2)$ est plus petit que $RejointD(S_3)$. 4 rounds plus tard (chaque flèche représente 2 rounds), la fusion a lieu.

**Lemme 7.4.7**
Soit $\mathcal{E}$ une exécution. Soit $S_2$ un privilège lent et $F_3$ un privilège rapide. Si $RejointDD(S_2) \leq RejointDD(F_3)$, alors $F_3$ disparaîtra avant de franchir $D$ deux fois.

**Preuve :** Même idée de preuve que celle du lemme 7.4.6. □

Une illustration de ce lemme est donné figure 7.5. Dans la configuration $C_0$, $RejointDD(S_2)$ est plus petit que $RejointDD(F_3)$. 10 rounds plus tard, la fusion a lieu.

**Lemme 7.4.8**
Soit $\mathcal{E}$ une exécution. Soit $S_2$ et $S_3$ deux privilèges lents. Si $RejointDD(S_3) \leq RejointD(S_2)$, alors $S_2$ disparaîtra avant de franchir $D$.

**Preuve :** Même idée de preuve que celle du lemme 7.4.6. □

Une illustration de ce lemme est donné figure 7.5. Dans la configuration $C_1$, $RejointDD(S_2)$ est plus petit que $RejointD(S_3)$. 8 rounds plus tard, la fusion a lieu.

**Lemme 7.4.9**
Soit $\mathcal{E}$ une exécution, $C_0$ sa configuration initiale et $F_1$, $S_2$ et $F_3$ trois privilèges. En moins de $3n$ rounds, $\mathcal{E}$ aura atteint une configuration dans laquelle au moins un des trois privilèges aura disparu.

**Preuve :** (par l'absurde) Supposons qu'aucun des trois privilèges ne disparaisse (en rattrapant ou en étant rattrapé) pendant les $3n$ rounds suivants $C_0$. Cela signifie que (lemme 7.4.6 et 7.4.7) :

$$RejointD(F_1) > RejointD(S_2)$$

$$RejointDD(S_2) > RejointDD(F_3)$$

Or, en utilisant la définition de $RejointDD$ (7.4.5), cette deuxième équation peut aussi s'écrire :

$$RejointD(S_2) + n > RejointD(F_3) + 2n$$

Au final, on obtient

$$RejointD(F_1) > RejointD(S_2) > RejointD(F_3) + n$$

ce qui est impossible car un privilège rapide a toujours un $RejointD$ plus petit que $n$ (et que $RejointD$ est une fonction positive). □

**Lemme 7.4.10**
Soit $\mathcal{E}$ une exécution, $C_0$ sa configuration initiale et $S_1$, $F_2$ et $S_3$ trois privilèges. En moins de $3n$ rounds, $\mathcal{E}$ aura atteint une configuration dans laquelle au moins un des trois privilèges aura disparu.

**Preuve :** (par l'absurde) Supposons qu'aucun des trois privilèges ne disparaisse pendant les $3n$ rounds suivant $C_0$. Cela signifie (lemme 7.4.6, 7.4.7 et 7.4.8) que

$$RejointDD(S_1) > RejointDD(F_2)$$

$$RejointD(F_2) > RejointD(S_3)$$

$$RejointDD(S_3) > RejointD(S_1)$$

Or, en utilisant la définition de $RejointDD$ (7.4.5), les équations 1 et 3 peuvent aussi s'écrire

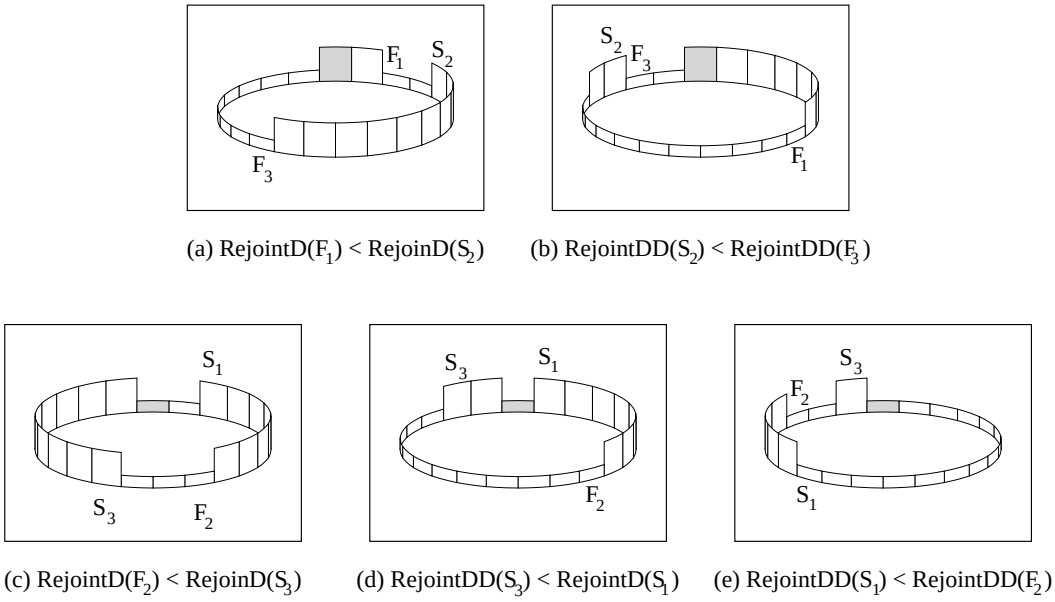
$$RejointD(S_1) + n > RejointD(F_2) + 2n$$

$$RejointD(S_3) + n > RejointD(S_1)$$

Au final, on obtient

$$RejointD(S_1)+n > RejointD(F_2)+2n > RejointD(S_3)+2n > RejointD(S_1)+n$$

ce qui est impossible car les inégalités sont strictes. □




(a) RejointD($F_1$) < RejoinD($S_2$) (b) RejointDD($S_2$) < RejointDD($F_3$)

(c) RejointD($F_2$) < RejoinD($S_3$) (d) RejointDD($S_3$) < RejointD($S_1$) (e) RejointDD($S_1$) < RejointDD($F_2$)

FIG. 7.6 – *Configurations types à 3 privilèges*

Des exemples illustrant les différents cas traités dans les lemmes 7.4.9 et 7.4.10 sont présentés figure 7.6. Une configuration à 3 privilèges vérifie nécéssairement la condition d'une des 5 configurations présentées.

**Remarque 7.4.11**

Les deux lemmes précédents assurent qu'au moins un des trois privilèges $S_1$,$F_2$,$S_3$ ou $F_1$,$S_2$,$F_3$ disparaît. Mais fort du principe que les privilèges disparaissent toujours par deux, il disparaît obligatoirement avec un des autres privilèges présents sur l'anneau.

**Lemme 7.4.12**
L'algorithme 7.1 converge.

**Preuve :** Soit $\mathcal{E}$ une exécution, $C_0$ sa configuration initiale et $2i+1$ le nombre initial de privilèges. On sait que la suite des privilèges présente une alternance de privilèges lents et rapides (propriété 7.3.1). Moins de $3n$ rounds après $C_0$, l'exécution atteint une configuration $C_1$ dans laquelle 2 privilèges ont disparu (lemme 7.4.9 ou 7.4.10). En itérant le processus, l'exécution atteint une configuration $C_i$ dans laquelle $2i$ privilèges ont disparu, et cela en moins de $3n \times i$ rounds. $C_i$ n'a qu'un seul privilège, c'est donc une configuration légitime. □

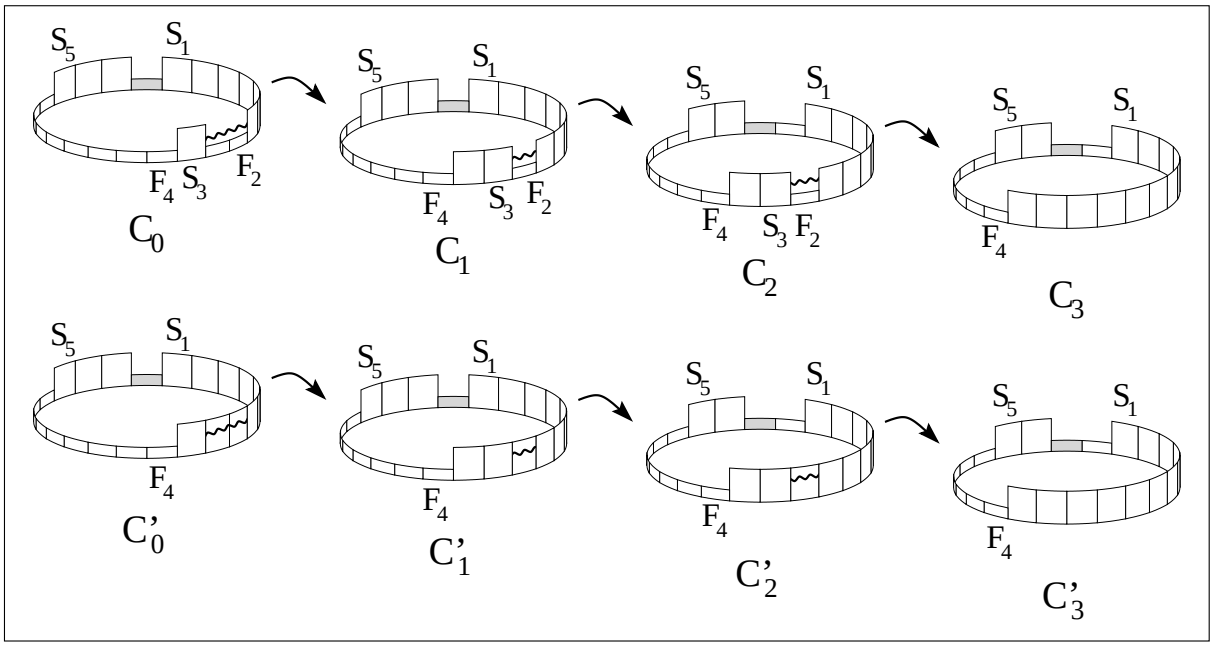



FIG. 7.7 – *Exécutions C-équivalentes*

### 7.4.4 Complexité

**Lemme 7.4.13**
Soit $\mathcal{E}$ une exécution et $C_0$ sa configuration initiale. Soit $C$ une configuration quelconque de $\mathcal{E}$ et $J$ un privilège de $C$. Alors $J$ existe dans $C_0$.

**Preuve :** Aucune des règles de l'algorithme ne permet à un processeur de créer un privilège. Un processeur ayant un privilège ne peut donc que l'avoir reçu de son prédécesseur. En itérant, si $J$ existe dans $C$, cela ne peut être que parce que des processeurs se le transmettent depuis le début de l'exécution. Il existe donc dans $C_0$. □

**Définition 7.4.14**
Soit $\mathcal{E}$ une exécution ayant $C$ pour configuration initiale. Soit $J_1$ et $J_2$ deux privilèges de $C$. On définit alors $C'$, la configuration obtenue à partir de $C$ en *supprimant* les privilèges $J_1$ et $J_2$, de la manière suivante :

1. Pour tout $P$ dans $[J_1..J_2[$, on pose $Priv_{C'}(P) = Priv_C(P) + 1$ ($modulo$ 2)
2. Pour tout $P$ dans $[J_2..J_1[$, on pose $Priv_{C'}(P) = Priv_C(P)$

On note $C' = C/\{J_1,J_2\}$ et on dit que $C'$ est obtenue en privant $C$ de $\{J_1,J_2\}$.
Par convention, si $J_1$ et $J_2$ n'existent pas dans $C$, $C/\{J_1,J_2\}$ est quand même défini et est égal à $C$.

Par exemple, la configuration $C'_0$ de la figure 7.7 est obtenue en supprimant $F_2$ et $S_3$ de $C_0$.

Au cours d'une exécution, quand deux privilèges $F_2$ et $S_3$ fusionnent, une configura-

tion $C$ est atteinte. Cette configuration $C$ peut également être atteinte par une exécution dans laquelle $F_2$ et $S_3$ n'existent pas. La figure 7.7 illustre cette propriété des exécutions : à partir de la configuration $C_0$ , l'exécution $\mathcal{E}$ conduit à la configuration $C_3$ dans laquelle $F_2$ et $S_3$ viennent de fusionner. L'exécution $\mathcal{E}'$, dont la configuration initiale $C'_0$ est obtenue en supprimant $F_2$ et $S_3$ de $C_0$, conduit également vers $C_3$.

Le lemme suivant est une généralisation de cet exemple.

**Lemme 7.4.15**
Soit $\mathcal{E} = \{C_0,C_1,C_2,...\}$ une exécution et $J_1$, $J_2$ les deux premiers privilèges de $\mathcal{E}$ à fusionner. Soit $C'_0$ la configuration obtenue en privant $C_0$ de $\{J_1,J_2\}$. Alors, si $\mathcal{E}' = \{C'_0,C'_1,C'_2,...\}$ est l'exécution ayant $C'_0$ pour configuration initiale, pour toute configuration $C'_i$ de $\mathcal{E}'$, on a $C'_i = C_i/\{J_1,J_2\}$

**Preuve : (par récurrence sur $i$)**

1. Si $i = 0$ : $C'_0 = C_0/\{J_1,J_2\}$, par définition.
2. Supposons que le lemme soit vrai pour $i-1$ (avec $i \geq 1$), c'est à dire : $C'_{i-1} = C_{i-1}/\{J_1,J_2\}$
3. Montrons que $C'_i = C_i/\{J_1,J_2\}$ :
   $C'_i$ est le but d'une transition ayant $C'_{i-1}$ pour origine. Soit $C''_i$ la configuration obtenue en privant $C_i$ de $\{J_1,J_2\}$. Montrons que $C'_i$ et $C''_i$ sont identiques. Pour cela, considérons un processeur $P$. Étudions la valeur de $Priv(P)$.
   $P$ peut avoir ou ne pas avoir de privilèges. De même, il peut être ou ne pas être dans l'intervalle $[J_1..J_2[$. On remarque que s'il est dans l'intervalle $[J_1..J_2]$ et qu'il est privilégié, on a $P = J_1$ ou $P = J_2$, car il n'y a pas de privilège entre $J_1$ et $J_2$ (car ils sont les premiers à fusionner). En fonction de ces paramètres, on a les égalités (ou inégalités) suivantes :
   (a) Si, dans $C_{i-1}$, $P$ n'est pas dans l'intervalle $[J_1..J_2[$ et ne transmet pas de privilège :
   Dans $C_{i-1}$, on a $P \notin [J_1..J_2[$ d'où : $Priv_{C_{i-1}}(P)=Priv_{C'_{i-1}}(P)$
   Dans $C_i$, on a également $P \notin [J_1..J_2[$ d'où : $Priv_{C_i}(P)=Priv_{C''_i}(P)$
   Dans $C_{i-1}$, $P$ ne transmet pas de privilège : $Priv_{C_{i-1}}(P)=Priv_{C_i}(P)$
   $P$ ne transmettant pas de privilège dans $C_{i-1}$, il n'en transmet pas non plus dans $C'_{i-1}$. D'où : $Priv_{C'_{i-1}}(P) = Priv_{C'_i}(P)$
   Au final, on a

$$Priv_{C'_i}(P) = Priv_{C'_{i-1}}(P) = Priv_{C_{i-1}}(P) = Priv_{C_i}(P) = Priv_{C''_i}(P)$$

   D'où $Priv_{C'_i}(P) = Priv_{C''_i}(P)$.
   (b) Si $P_{C_{i-1}} \in [J_1..J_2[$ et $P$ ne transmet pas de privilège, une étude similaire montre que

| | | | |
|---|---|---|---|
| $Priv_{C_{i-1}}(P)$ | $\neq$ | $Priv_{C'_{i-1}}(P)$ | car dans $C_{i-1}$, $P \in [J_1..J_2[$ |
| $Priv_{C_i}(P)$ | $\neq$ | $Priv_{C''_i}(P)$ | car dans $C_i$, $P \in [J_1..J_2[$ |
| $Priv_{C_{i-1}}(P)$ | $=$ | $Priv_{C_i}(P)$ | car dans $C_{i-1}$, $P$ n'est pas privilégié |
| $Priv_{C'_{i-1}}(P)$ | $=$ | $Priv_{C'_i}(P)$ | car dans $C'_{i-1}$, $P$ n'est pas privilégié |

Au final, on a

$$Priv_{C'_i}(P) = Priv_{C'_{i-1}}(P) \neq Priv_{C_{i-1}}(P) = Priv_{C_i}(P) \neq Priv_{C''_i}(P)$$

Comme $Priv$ ne peu prendre que deux valeurs et que deux variables booléennes différentes d'une même troisième sont égales entre elle, on a bien $Priv_{C'_i}(P) = Priv_{C''_i}(P)$.

(c) Si $P_{C_{i-1}} \notin [J_1..J_2]$ et $P$ transmet un privilège, on obtient :

$$\begin{array}{rcll} Priv_{C_{i-1}}(P) & = & Priv_{C'_{i-1}}(P) & \text{car dans } C_{i-1},\ P \notin [J_1..J_2] \\ Priv_{C_i}(P) & = & Priv_{C''_i}(P) & \text{car dans } C_i,\ P \notin [J_1..J_2[ \\ Priv_{C_{i-1}}(P) & \neq & Priv_{C_i}(P) & \text{car dans } C_{i-1},\ P \text{ est privilégié} \\ Priv_{C'_{i-1}}(P) & \neq & Priv_{C'_i}(P) & \text{car dans } C'_{i-1},\ P \text{ est privilégié} \end{array}$$

Au final, on a

$$Priv_{C'_i}(P) \neq Priv_{C'_{i-1}}(P) = Priv_{C_{i-1}}(P) \neq Priv_{C_i}(P) = Priv_{C''_i}(P)$$

D'ou $Priv_{C'_i}(P) = Priv_{C''_i}(P)$.

(d) Si $P_{C_{i-1}} \in [J_1..J_2]$ et $P$ transmet le privilège $J_1$ :

$$\begin{array}{rcll} Priv_{C_{i-1}}(P) & \neq & Priv_{C'_{i-1}}(P) & \text{car dans } C_{i-1},\ P \in [J_1..J_2[ \\ Priv_{C_i}(P) & = & Priv_{C''_i}(P) & \text{car dans } C_i,\ P \notin [J_1..J_2[ \\ Priv_{C_{i-1}}(P) & \neq & Priv_{C_i}(P) & \text{car dans } C_{i-1},\ P \text{ est privilégié} \\ Priv_{C'_{i-1}}(P) & = & Priv_{C'_i}(P) & \text{car dans } C'_{i-1},\ J_1 \text{ n'existe pas} \end{array}$$

Au final, on a

$$Priv_{C'_i}(P) = Priv_{C'_{i-1}}(P) \neq Priv_{C_{i-1}}(P) \neq Priv_{C_i}(P) = Priv_{C''_i}(P)$$

D'ou $Priv_{C'_i}(P) = Priv_{C''_i}(P)$.

(e) Si $P_{C_{i-1}} \in [J_1..J_2]$ et $P$ transmet le privilège $J_2$ :

$$\begin{array}{rcll} Priv_{C_{i-1}}(P) & = & Priv_{C'_{i-1}}(P) & \text{car dans } C_{i-1},\ P \notin [J_1..J_2[ \\ Priv_{C_i}(P) & \neq & Priv_{C''_i}(P) & \text{car dans } C_i,\ P \in [J_1..J_2[ \\ Priv_{C_{i-1}}(P) & \neq & Priv_{C_i}(P) & \text{car dans } C_{i-1},\ P \text{ est privilégié} \\ Priv_{C'_{i-1}}(P) & = & Priv_{C'_i}(P) & \text{car dans } C'_{i-1},\ J_2 \text{ n'existe pas} \end{array}$$

Au final, on a

$$Priv_{C'_i}(P) = Priv_{C'_{i-1}}(P) = Priv_{C_{i-1}}(P) \neq Priv_{C_i}(P) \neq Priv_{C''_i}(P)$$

D'ou $Priv_{C'_i}(P) = Priv_{C''_i}(P)$.

Dans tous les cas, $P$ a la même valeur dans $C'_i$ et dans $C''_i$. D'ou $C'_i = C''_i$ et on a bien $C'_i = C_i/\{J_1,J_2\}$.

Au final, la propriété de récurrence est vraie au rang 0 et la supposer vraie au rang $i-1$ permet de la montrer au rang $i$. Elle est donc vrai pour tout $i$. □

**Lemme 7.4.16**
Pour toute exécution ayant $2i+1$ privilèges dans sa configuration initiale (avec $i \geq 2$), il existe une exécution C-équivalente dont la configuration initiale n'a que $2i-1$ privilèges.

**Preuve :** Soit $\mathcal{E}$ une exécution ayant au moins 5 privilèges dans sa configuration initiale $C_0$. Soit $J_1$ et $J_2$ les deux premiers privilèges qui fusionnent lors

de l'exécution $\mathcal{E}$. Montrons que l'exécution $\mathcal{E}'$ dont la configuration initiale est $C'_0 = C_0/\{J_1,J_2\}$ est C-équivalente à $\mathcal{E}$.

Pour toute configuration $C'_i$ de $\mathcal{E}'$, on a $C'_i = C_i/\{J_1,J_2\}$ (lemme 7.4.15). Après la disparition de $J_1$ et $J_2$ cela est toujours vrai. Cela est encore vrai pour la première configuration légitime de $\mathcal{E}$ qui est également une configuration légitime de $\mathcal{E}'$. Les deux exécutions sont donc bien C-équivalentes. □

**Lemme 7.4.17**
Pour toute exécution ayant initialement 3 privilèges ou plus, il existe une exécution C-équivalente dont la configuration initiale n'a que trois privilèges.

**Preuve :** Soit $\mathcal{E}$ une exécution ayant initialement $2i+1$ privilèges avec $i \geq 2$ (si $i = 1$, la propriété est immédiate). Il existe $\mathcal{E}_i$ une exécution C-équivalente à $\mathcal{E}$ ayant initialement $2i-1$ privilèges (lemme 7.4.16). De même, il existe $\mathcal{E}_{i-1}$ une exécution C-équivalente à $\mathcal{E}_i$ ayant initialement $2i-3$ privilèges. De proche en proche, on établit l'existance de $\mathcal{E}_1$ une exécution C-équivalente à $\mathcal{E}_2$ ayant initialement 3 privilèges. La relation C-équivalente étant transitive (lemme 2.3.8), $\mathcal{E}$ est C-équivalente a $\mathcal{E}_1$ □

**Lemme 7.4.18**
L'algorithme 7.1 converge en $O(n)$ rounds.

**Preuve :** Toute exécution $\mathcal{E}$ est C-équivalente à une exécution $\mathcal{E}'$ ayant initialement trois privilèges (lemme 7.4.17). Or, après au plus $3n$ rounds, un des trois privilèges de $\mathcal{E}'$ disparaît (lemme 7.4.9 ou 7.4.10). Comme les privilèges disparaissent toujours par deux, deux des trois privilèges disparaissent et $\mathcal{E}'$ converge en moins de $3n$ rounds.

$\mathcal{E}$ étant C-équivalente à $\mathcal{E}'$, elle converge également en moins de $3n$ rounds. D'où la complexité en temps de l'algorithme est $O(n)$ □

### 7.4.5 Bilan

Tous les éléments nécessaires à la preuve du théorème 7.2.1 sont maintenant réunis.

**Théorème 7.4.1**
L'algorithme présenté figure 7.1 est auto-stabilisant pour le problème de l'exclusion mutuelle. Si la taille de l'anneau est $n$, sa mise en application nécessite 1 bit par processeur et son temps de stabilisation est de $O(n)$.

**Preuve :** L'ensemble des configurations légitimes $\mathcal{L}$ (défini au 7.4.1) vérifie la correction (lemme 7.4.4) et la convergence (lemme 7.4.12). De plus, sa phase de stabilisation est de l'ordre de $n$ rounds (lemme 7.4.18). D'où le système est auto-stabilisant pour l'exclusion mutuelle et son temps de convergence est $O(n)$ rounds.

L'implémentation de l'algorithme nécessite pour chaque processeur une unique variable booléenne. Sa complexité en espace est donc de 1 bit. □

# Chapitre 8

# Exclusion mutuelle synchrone proportionnelle

L'algorithme que nous avons présenter au chapitre précédent -nous l'appellerons "algorithme de base synchrone"- est auto-stabilisant mais son temps de convergence en cas de faible corruption peut être élevé. En particulier, une simple faute peut se propager à travers tout le réseau, entraînant la perturbation de nombreux processeurs et un temps de retour à la normale élevé. Dans cette section, nous présentons un nouvel algorithme assurant une stabilisation proportionnelle.

## 8.1 Retour à l'algorithme de base synchrone

**Problème :** Considérons l'algorithme de base synchrone. Après une corruption de faible envergure, un certain nombre de privilèges sont présents sur l'anneau. Les propriétés présentées dans la section précédente (propriétés 7.3.1, page 123) nous assurent qu'il a y alternance entre les privilèges rapides et les lents. D'autre part, nous avons vu au chapitre 3 qu'en cas de corruption de $f$ processeurs, un privilège dû à une corruption a toujours un autre privilège parmi ses $f$ successeurs ou ses $f$ prédécesseurs.

Dans le cas synchrone, deux cas peuvent se produire :

1. **C1 :** Un privilège rapide a parmi ses $f$ successeurs un privilège lent : le privilège rapide avance tous les rounds, le lent tous les deux rounds. La distance entre les deux privilèges diminue donc progressivement jusqu'à ce que les deux processeurs fusionnent. La "poursuite" dure $2f$ rounds.
2. **C2 :** Un privilège lent a parmi ses $f$ successeurs un privilège rapide : à chaque round, la distance entre les deux privilèges s'accroît, et une convergence rapide n'est pas possible.

La figure 8.1 illustre les deux cas possibles. Les configurations $C_1$ et $C'_1$, toutes deux obtenues à partir d'une configuration légitime en corrompant un unique processeur, sont les configurations initiales des exécutions $\mathcal{E}$ et $\mathcal{E}'$. $\mathcal{E}$ converge rapidement, car $F_1$ rattrape $S_2$ en deux rounds. Par contre, dans $\mathcal{E}'$, $F_2$ s'éloigne de plus en plus de $S_1$ et la convergence est proportionnelle à la taille de l'anneau.

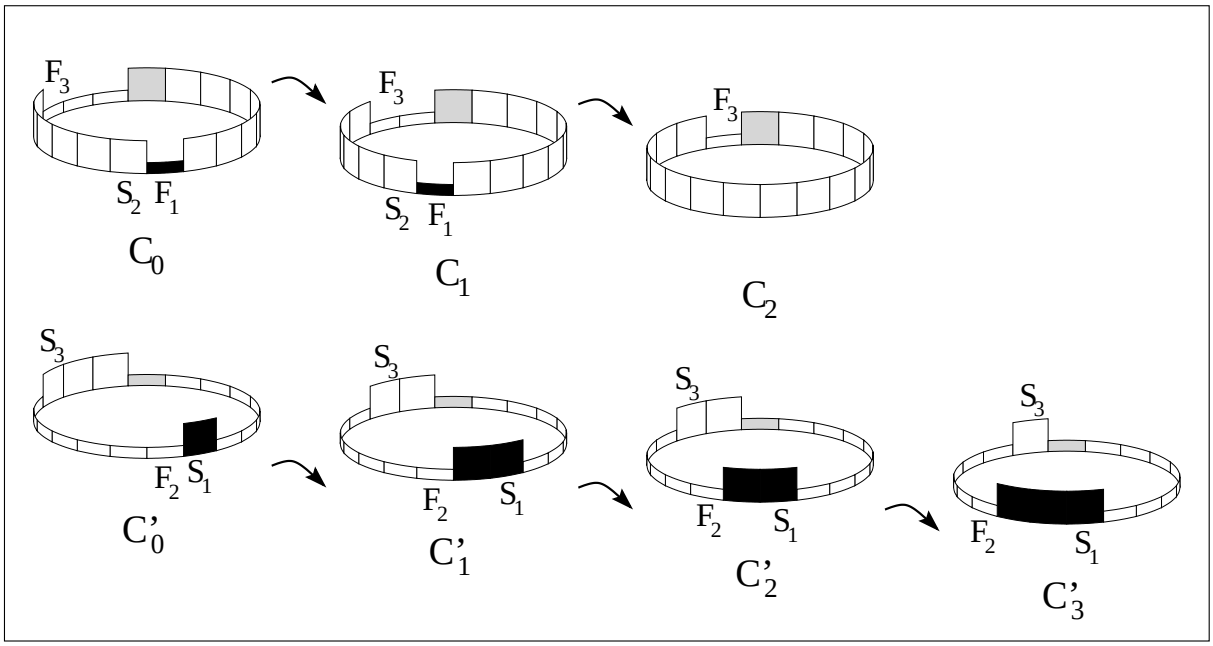



FIG. 8.1 – *Une faute : deux cas possibles*

## 8.2 Solution utilisant la procédure $TEST$

Cette situation n'est pas sans rappeler celle rencontrée dans le chapitre précédent. Les corruptions faisaient alors apparaître de faux privilèges et des privilèges correcteurs. En avançant, les uns éloignaient la configuration d'une configuration légitime, les autres l'en rapprochaient. La solution fut alors d'interdire tout mouvement aux faux privilèges, ce qui nous permit d'obtenir un temps de convergence de $O(k^2n)$ transitions. Dans le cas présent, l'utilisation du même genre de techniques aboutit à un algorithme d'exclusion mutuelle auto-stabilisant et anti-corruption avec un temps de convergence de l'ordre de $O(k^2)$ rounds.

Le chapitre 3 présente un algorithme ayant un temps de convergence de $O(k^2n)$ transitions sous un démon central. La même technique appliquée au cas synchrone permet d'obtenir un algorithme convergeant en $O(k^2)$ rounds. Cette différence de temps de convergence pourrait laisser penser que le deuxième algorithme est plus efficace que le premier et à ce titre constitue une amélioration. Il n'en est rien. En effet, dans un round, le démon synchrone fait agir $n$ processeurs alors que dans une transition, le démon central n'en fait agir qu'un. Les performances des deux algorithmes sont donc comparables.

Mais au delà de la procédure $TEST$, la nature synchrone du présent démon nous permet néanmoins d'améliorer ces performances, notamment grâce aux notions de propagation d'information et de vitesses relatives.

## 8.3 Techniques d'amélioration synchrones

**Blocage des faux privilèges rapides :** Sur les deux cas possibles de corruption (**C1** et **C2**), seul le deuxième est problématique. Dans le cas **C1**, l'algorithme converge rapidement. Il n'y a donc pas lieu de le modifier. Dans le cas **C2**, le privilège rapide avance alors qu'il serait plus intéressant qu'il reste sur place en attendant de se faire rattraper.

D'un point de vue pratique, le problème se résume donc à permettre aux privilèges rapides de distinguer s'ils sont dans le cas **C1** ou **C2**. Or, une des différences majeures entre les deux cas est que dans le premier, le privilège rapide est plus proche de son privilège successeur que de son privilège prédécesseur alors que dans le second, c'est l'inverse qui se produit.

Dès lors, une solution consiste à mettre à disposition des privilèges rapides des informations concernant la distance les séparant de leur privilège prédécesseur et de leur privilège successeur. Forts de ces informations, ils adoptent alors le comportement suivant :

1. Si le privilège rapide est plus proche de son privilège successeur que de son privilège prédécesseur, il avance rapidement pour rattraper au plus tôt son privilège successeur.
2. Dans le cas contraire, il ne bouge pas car il sait qu'il fusionnera plus vite en attendant d'être rattrapé.

Appliqué au cas **C1**, le nouveau comportement du privilégié est identique à l'ancien : les informations dont il dispose lui indiquent qu'un privilège est présent à une courte distance devant lui. Il avance donc pour le rattraper. Par contre, dans le cas **C2**, le privilège rapide est informé qu'un privilège lent est présent parmi ses proches prédécesseurs. Il décide donc de ne plus avancer et de se faire rattraper. Dans tous les cas, le privilégié choisit la solution qui mènera le plus rapidement à la convergence.

**Distance entre privilèges** Reste à fournir aux privilèges rapides des informations sur la distance les séparant de leurs privilèges voisins. Cette opération se fait grâce à deux variables dédiées, l'une à la distance au successeur, l'autre à la distance au prédécesseur.

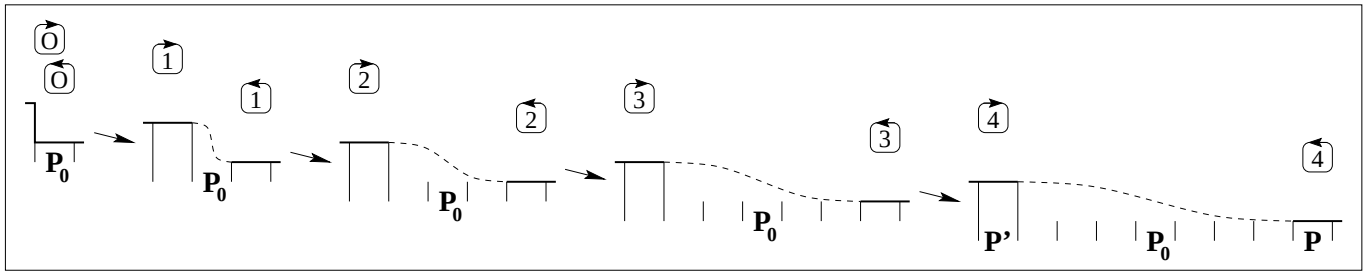
A la base, un processeur privilégié indique qu'il a un privilège en positionnant ses variables distances à zéro. Son successeur positionne sa variable à 1, puis le successeur du successeur positionne la sienne à 2, et ainsi de suite. De proche en proche, l'information se propage, permettant ainsi à un processeur de savoir à quelle distance il se trouve de son privilège prédécesseur.

Naturellement, le même procédé est utilisé pour le privilège successeur.

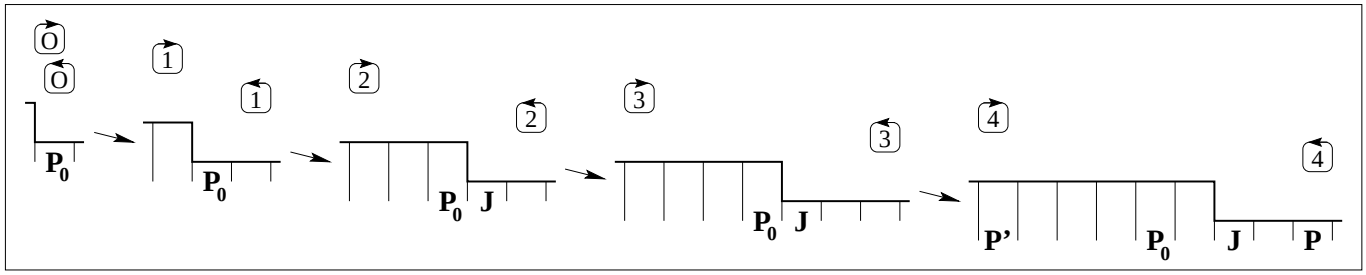
La figure 8.2.(a) illustre le principe.

**Vitesses relatives** Comme nous l'avons déjà précisé, la propagation de l'information se doit d'être plus rapide que le mouvement des privilèges. Aussi, l'algorithme considère trois types de vitesses différentes :

1. Tous les rounds, les variables distances sont réactualisées.
2. Tous les deux rounds, les privilèges rapides actifs sont transmis.




(a) Propagation des informations sur les distances




(b) Vitesse relative des indices et des privilèges

FIG. 8.2 – *Variables distances : mythes et réalités*

3. Tous les quatre rounds, les privilèges lents bougent.

Ainsi, la propagation d'information se fait plus rapidement que le mouvement des privilèges et les privilèges gardent des vitesses relatives différentes (un privilège rapide actif avance deux fois plus vite qu'un privilège lent, comme dans l'algorithme d'exclusion mutuelle synchrone).

**Distance réelle :** dernier détail, la propagation des informations n'est pas instantanée : ainsi, lorsqu'un processeur a une information de distance, elle peut être rendue obsolète par les mouvements du privilège. La figure 8.2 illustre le phénomène. Dans cet exemple, la variable contenant l'information sur le privilège prédécesseur est dans un cadre surmonté d'une flèche vers la droite (le cadre le plus bas) alors que celle concernant le privilège successeur est dans un cadre surmonté d'une flèche vers la gauche (le cadre le plus haut). Initialement, $P_0$ a un privilège $J$. Quand les processeurs $P'$ et $P$ reçoivent les informations sur leur distance à $J$, celui-ci a bougé et les informations sont obsolètes.

Néanmoins, si un processeur $P$ connaît la nature de son privilège prédécesseur $J^-$, il lui est possible d'évaluer la distance que $J^-$ a pu parcourir pendant que l'information lui parvenait. Ainsi, quand $P$ a pour information que $J^-$ est à distance $d$, il sait que cette information a mis $d$ rounds pour lui parvenir. Aussi, selon la nature de $J^-$, il effectue une correction :

1. Si $J^-$ est un privilège rapide actif : en $d$ rounds, il a été transmis $d/2$ fois[1]. La

1. Pour être tout a fait formel, si $d$ est paire, $J^-$ a été transmis $d/2$ fois alors que si $d$ est impair, $J^-$ a été transmis $(d-1)/2$ fois ou $(d+1)/2$ fois. Mais cette précision de chiffre alourdi considérablement les notations sans rien apporter aux explications ou aux preuves. Aussi continuerons nous à parler de

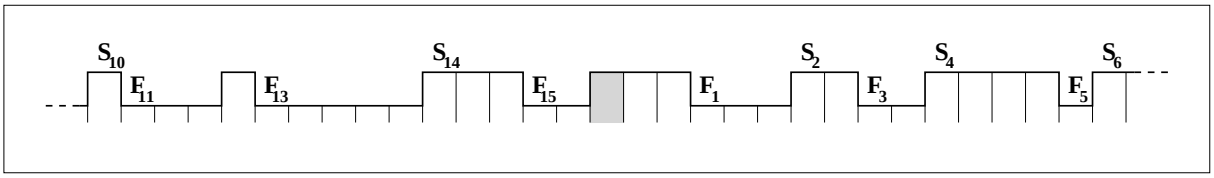



FIG. 8.3 – *Voisins des privilèges rapides*

distance réelle entre $J^-$ et $P$ est donc de $d - d/2$.

2. Si $J^-$ est un privilège lent : en $d$ rounds, il a été transmis $d/4$ fois[2]. La distance réelle entre $J^-$ et $P$ est donc de $d - d/4$.
3. Si $J^-$ est un privilège rapide inactif : en $d$ rounds, il n'a pas été transmis. La distance réelle entre $J^-$ et $P$ est donc de $d$.

De même, si $P'$ a pour information que son privilège successeur $J^+$ est à distance $d'$ :

1. Si $J^+$ est un privilège rapide actif : en $d'$ rounds, il a été transmis $d'/2$ fois. La distance réelle entre $P'$ et $J^+$ est donc de $d' + d'/2$.
2. Si $J^+$ est un privilège lent : en $d'$ rounds, il a été transmis $d'/4$ fois. La distance réelle entre $P'$ et $J^+$ est donc de $d' + d'/4$.
3. Si $J^+$ est un privilège rapide inactif : en $d'$ rounds, il n'a pas été transmis. La distance réelle entre $P'$ et $J^+$ est donc de $d'$.

Un processeur peut donc évaluer la distance le séparant d'un privilège dont il connaît la nature.

**Nature des privilèges voisins :** Nous l'avons déjà remarqué précédemment (propriété 7.3.1, page 123), la suite des privilèges présents sur l'anneau présente une alternance de privilèges rapides et de privilèges lents. Cette propriété va être determinante dans l'évaluation des distances entre processeurs.

En effet, si tous les processeurs disposent de l'information distance, seuls les possesseurs d'un privilège rapide en ont l'usage. Cela leur permet de rendre leur privilège actif ou inactif. Or, dans la majorité des cas, un privilège rapide a pour privilèges voisins des privilèges lents. Il connaît donc la distance exacte le séparant de ses privilèges voisins. Par exemple, s'il dispose d'une information lui indiquant un privilège prédécesseur à distance $d$, il sait que la distance réelle l'en séparant est $d - d/4$.

**Exception :** Dans la pargraphe précédent, nous avons supposé qu'un privilège rapide a pour privilèges voisins des privilèges lents. Même si cela est vrai pour la majorité des privilèges rapides, cela n'est pas toujours le cas. En effet, si $D$ a pour privilège successeur un privilège rapide $F_1$, il a également un privilège rapide $F_{15}$ pour privilège prédécesseur, et $F_1$ $F_{15}$ sont privilèges voisins l'un de l'autre comme le montre la figure 8.3 (où tous les privilèges rapides ont pour privilèges voisins des privilèges lents sauf $F_1$ et $F_{15}$). Il en résulte que le mécanisme de correction des distances présenté précédemment

$d/2$ (et plus tard de $d/4$) là où il aurait fallu distinguer deux cas, selon que la division de $d$ par 2 (ou par 4) est entière ou non.

2. Voir remarque précédente.

donne un résultat faux. Mais cette erreur d'appréciation de $F_1$ et $F_{15}$ ne prête pas à conséquence. En effet, elle n'empêche pas la convergence, pas plus qu'elle n'entraîne de surcoût significatif[3] en terme de temps de convergence. Aussi nous garderons nous de modifier l'algorithme.

## 8.4 Hypothèses & résultats

Les hypothèses de cet algorithme sont les mêmes que celle du précédent : toujours dans le **modèle à état**, le graphe considéré est un **anneau semi-uniforme bidirectionnel orienté** et le démon est un **démon synchrone**.

Sous ces hypothèses, nous proposons un algorithme d'exclusion mutuelle proportionnel utilisant $O(Log(n))$ bits de mémoire.

**Théorème 8.4.1**
L'algorithme présenté figure 8.4 est proportionnel pour le problème de l'exclusion mutuelle. De plus, si $n$ est la taille de l'anneau et $f$ le nombre initial de fautes, sa mise en application nécessite $O(Log(n))$ bits par processeur, il converge en $O(f^3)$ rounds pour $f \leq \sqrt[3]{n}$ et en $O(nLog(n))$ rounds pour $f > \sqrt[3]{n}$.

## 8.5 Algorithme

L'algorithme est présenté figure 8.4.

Chaque processeur $P$ dispose d'une variable $Priv(P)$ déterminant la position du privilège. $Priv(P)$ est à valeurs dans [0..1]. Les opérations d'incrémentation de $Priv(P)$ se font modulo 2. $P$ dispose aussi de deux variables $DistPred(P)$ et $DistSucc(P)$. Pour un processeur $P$ non privilégié, $DistPred(P)$ est la distance séparant $P$ de son privilège prédécesseur, alors que $DistSucc(P)$ est la distance séparant $P$ de son privilège successeur. Pour un processeur privilégié, ces deux variables sont à zéro (mais un privilégié peut néanmoins connaître la distance le séparant de ces privilèges voisins en accedant aux variables distances de ces voisins.

*DistSucc* et *DistPred* sont regroupés sous le terme générique de *variables distances*. Elles sont à valeur dans $[0..n]$.

**Prédicat :** le processeur distingué $D$ a le privilège si sa variable $Priv(D)$ a la même valeur que celle de son prédécesseur alors qu'un processeur $P$ non distingué est privilégié quand $Priv(P)$ est différent de $Priv(P^-)$.

3. Un algorithme prenant cette erreur en compte[4]aurait un temps de convergence au pire 1,5 fois plus court que l'algorithme actuel, c'est à dire un temps de convergence ayant le même ordre de grandeur.

4. Pour obtenir un tel algorithme, on peut par exemple adjoindre à chaque information sur les distances la nature du privilège les émettant. Ainsi, plutôt que de propager des nombres 0, 1, 2, les variables distances propageraient $0_F$, $1_F$, $2_F$ pour un privilège rapide actif, $0_S$, $1_S$, $2_S$ pour un privilège lent et $0_I$, $1_I$, $2_I$ pour un privilège rapide inactif. Mais cet ajout compliquerait inutilement l'algorithme sans en améliorer les performances[9].

Un privilège est rapide si la variable $Priv$ de son support a pour valeur 0. Sinon, il est lent.

Un processeur est actif s'il est plus proche de son privilège successeur que de son privilège prédécesseur (dans le cas contraire, il est inactif). Ce prédicat n'est utilisé que par les privilèges rapides. Or la distance d'un privilège rapide $F_2$ a son privilège prédécesseur $S_1$ est $Dist(S_1,F_2) - Dist(S_1,F_2)/4$ alors que sa distance à son privilège successeur $S_3$ est $Dist(F_2,S_3) + Dist(F_2,S_3)/4$ (comme cela a été specifié dans la section 8.3, paragraphe **distance réelle**).

**Actions :** trois types d'actions sont possibles. Tous les processeurs doivent régulièrement mettre à jour le contenu de leur variable $DistPred$ et $DistSucc$. C'est le résultat des actions $ActualisePred(P)$ et $ActualiseSucc(P)$. On remarque que l'action $ActualiseSucc(P)$ utilise le fait que $P$ puisse determiner si $P^+$ est privilégié. En effet, si la valeur de $Priv(P^+)$ est différente de celle de $Priv(P)$, $P^+$ a un privilège. Comme l'anneau est bidirectionnel, $P$ a accès à cette information.

Derniere action possible, un processeur privilégié peut transmettre son privilège. Pour cela, il modifie la valeur de sa variable $Priv$. C'est l'action $TransmetPrivilege(P)$.

**Règles :** la première règle gère l'actualisation des variables de distance. Elle est exécutée tous les rounds. Pour un processeur privilégié, elle consiste a mettre ses variables distances à zéro. Un processeur non privilégié $P$ met dans sa variable $DistSucc(P)$ la valeur de la variable $DistSucc(P^+)$ de son successeur *plus* un (même chose pour $DistPred$ avec le prédécesseur).

La deuxième règle n'est exécutée que tous les deux rounds par les processeurs disposant d'un privilège rapide plus proche de son privilège successeur que de son privilège prédécesseur. Dans ce cas, le processeur transmet son privilège.

La troisième et dernière règle est appliquée par les privilèges lents. Tous les quatre rounds, ils avancent.

## 8.6 Exemple d'exécutions

La figure 8.5 montre l'évolution des variables distances au cours d'une exécution légitime. Un privilège est transmis quand il est surmonté d'une flèche. Un privilège rapide contruit sa flèche en 2 rounds alors qu'un privilège lent construit la sienne en 4 rounds. On remarque qu'après un cycle (2 rounds pour les privilèges rapides et 4 rounds pour les privilèges lents), la valeur des indices est la même que la configuration initiale (relativement à la position du privilège).

La figure 8.6 illustre l'art et la manière qu'ont les processeurs ayant un privilège rapide de décider si leur privilège est un privilège actif ou inactif.

**Prédicats**

$Privilege(P) \Leftrightarrow \begin{cases} Priv(P) = Priv(P^-) & \text{si } P \text{ est le processeur distingué.} \\ Priv(P) \neq Priv(P^-) & \text{si } P \text{ est un processeur quelconque.} \end{cases}$

$PrivilegeRapide(P) \Leftrightarrow Privilege(P)$ et $Priv(P) = 0$

$PrivilegeLent(P) \Leftrightarrow Privilege(P)$ et $Priv(P) = 1$

$Actif(P) \Leftrightarrow DistSucc(P^+) \times (1 + \frac{1}{4}) \leq DistPred(P^-) \times (1 - \frac{1}{4})$ ou $DistPred(P^-) = n$

**Actions**

$ActualisePred(P) \Leftrightarrow \begin{cases} DistPred(P) \longleftarrow 0 & \text{Si } P \text{ est privilégié.} \\ DistPred(P) \longleftarrow DistPred(P^-) + 1 & \text{Sinon} \end{cases}$

$ActualiseSucc(P) \Leftrightarrow \begin{cases} DistSucc(P) \longleftarrow 0 & \text{Si } P \text{ est privilégié.} \\ DistSucc(P) \longleftarrow 1 & \text{Si } P^+ \text{ est privilégié.} \\ DistSucc(P) \longleftarrow DistSucc(P^+) + 1 & \text{Sinon} \end{cases}$

$ActualiseDist(P) \Leftrightarrow ActualisePred(P)$ et $ActualiseSucc(P)$

$TransmetPrivilege(P) \Leftrightarrow Priv(P) \longleftarrow Priv(P) + 1 (modulo\ 2)$

**Règles gardées**

R1 : Tous les rounds : $\Longrightarrow ActualiseDist(P)$

R2 : Tous les deux rounds : $PrivilegeRapide(P)$ et $Actif(P)$ $\Longrightarrow TransmetPrivilege(P)$

R3 : Tous les quatre rounds : $PrivilegeLent(P)$ $\Longrightarrow TransmetPrivilege(P)$

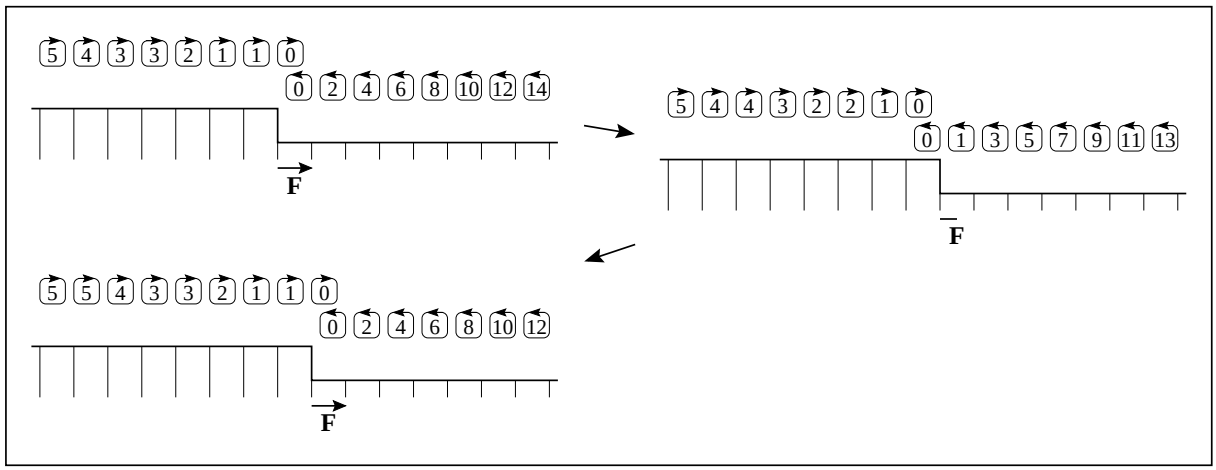
FIG. 8.4 – *Exclusion mutuelle proportionnelle*

## 8.7 Preuve

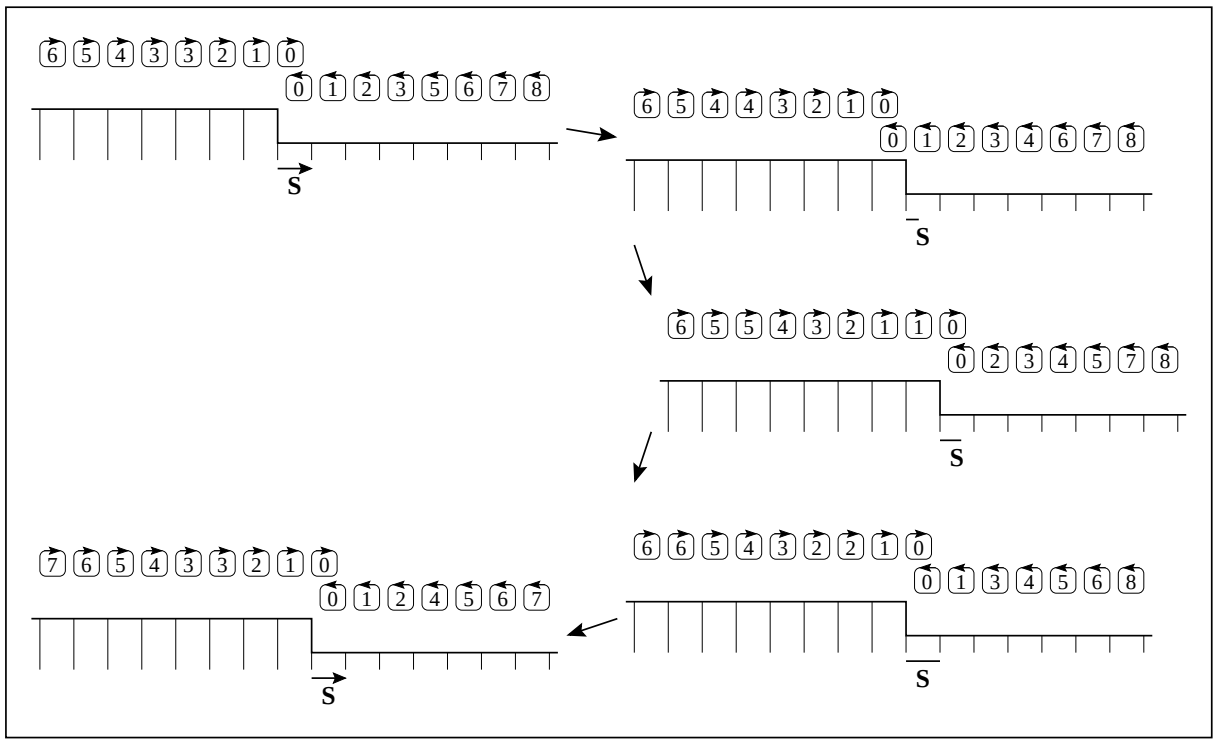
**Survol de la preuve :** Après avoir défini les configurations légitimes, nous montrons la correction puis la convergence. La preuve de la convergence se fait en établissant qu'après un certain nombre de rounds, les informations données pas les variables distances sont significatives (elles contiennent effectivement des informations sur les distances entre les différents processeurs privilégiés). Puis nous montrons qu'à partir d'une configuration où les variables distances sont correctes, une fusion de privilège doit nécessairement se produire. Ainsi, après une alternance de phases "mise à jour des variables distances" - "fusion de privilège", l'algorithme converge.

### 8.7.1 Configurations légitimes

**Définition 8.7.1**
Les configurations légitimes sont les configurations dans lesquelles un unique privilège est présent.




(a) Variables distances au voisinnage d'un privilège rapide




(b) Variables distances au voisinnage d'un privilège lent

FIG. 8.5 – *Evolution des variables indices*

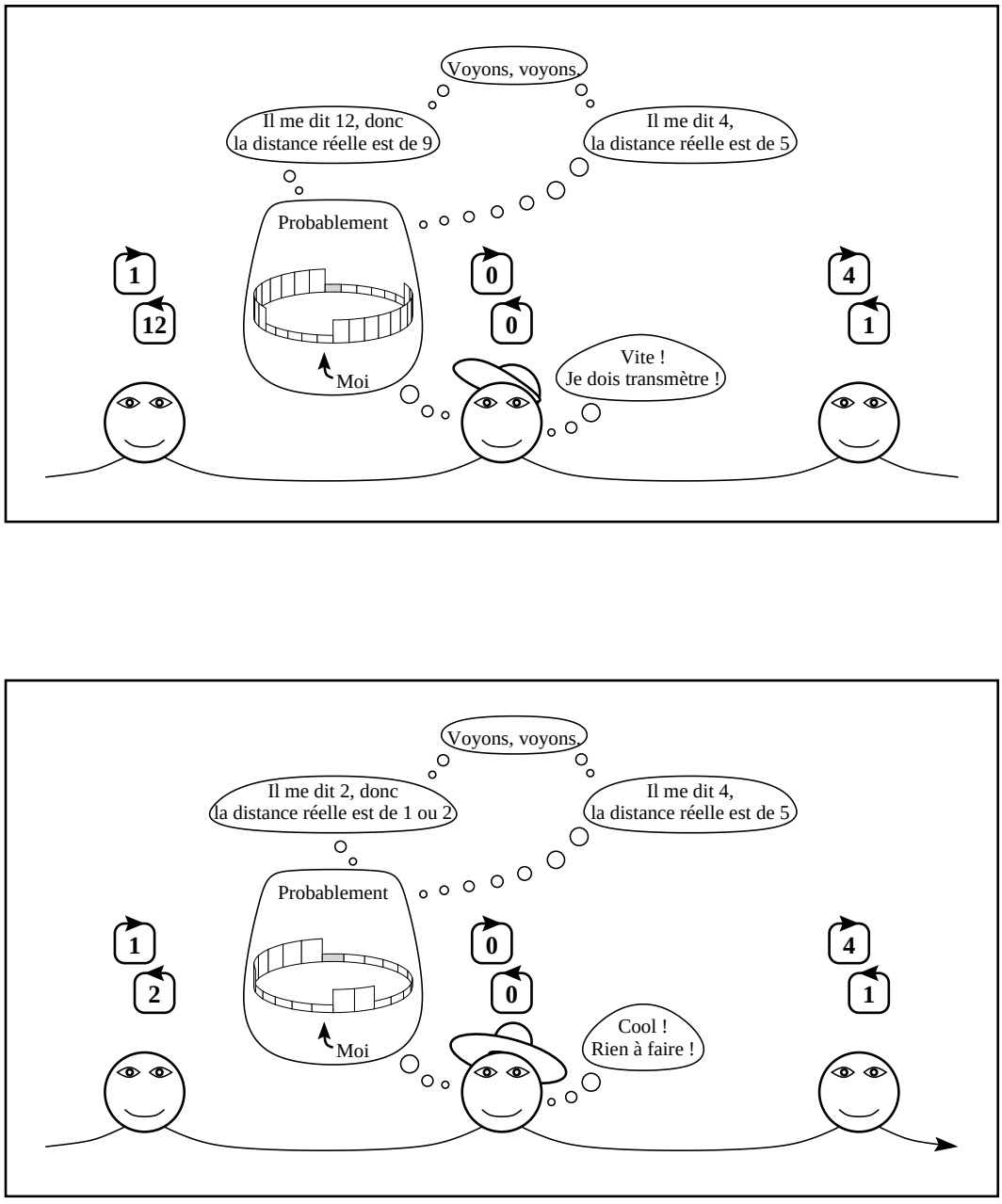



FIG. 8.6 – *Choix du privilège rapide*

**Définition 8.7.2**
Soit $T = C_1 \longrightarrow C_2$ une transition. On définit la notion de variable $DistPred(P)$ *correcte* recursivement :

1. Si $P$ possède un privilège, alors $DistPred(P)$ est correcte dans $C_2$ si elle est nulle.
2. Si $P$ n'est pas privilégié, alors $DistPred(P)$ est correcte dans $C_2$ si $DistPred(P^-)$ est correcte dans $C_1$ et que $DistPred_{C_2}(P)$ vaut $DistPred_{C_1}(P^-)$ plus un.

De la même manière :

1. Si $P$ possède un privilège, alors $DistSucc(P)$ est correcte dans $C_2$ si elle est nulle.
2. Si $P$ n'est pas privilégié, alors $DistSucc(P)$ est correcte dans $C_2$ si $DistSucc(P^+)$ est correcte dans $C_1$ et que $DistPred_{C_2}(P)$ vaut $DistPred_{C_1}(P^+)$ plus un.

### 8.7.2 Correction

**Survol de la preuve :** dans toutes les configuration d'une exécution légitime, un seul privilège est présent (car aucune règle ne permet la création de privilége). Il reste a établir que ce privilège est régulièrement transmis de processeur en processeur. Pour cela, nous montrons qu'aprés un certain temps, les variables indices sont correctes. Aussi, un processeur privilègié aura pour information que son privilège prédécesseur est trés loin et qu'il doit donc au plus vite transmettre son privilège.

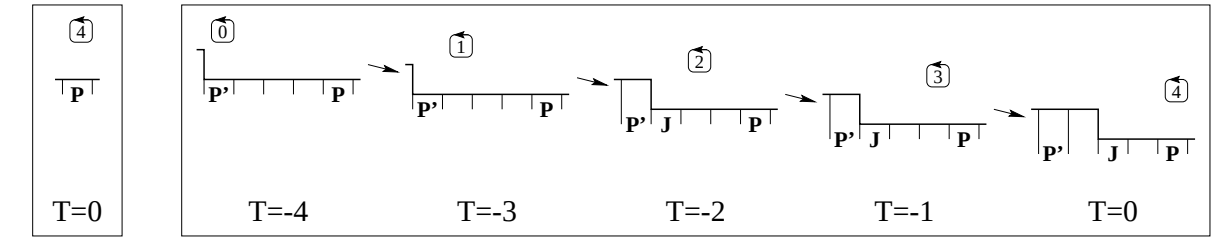
**Lemme 8.7.3**
Soit $\mathcal{E}$ une exécution, $C_i$ sa $i^{ieme}$ configuration, $P$ un processeur non privilégié et $J$ son privilège prédécesseur. Si $P'$ était le support de $J$ $DistPred(P)$ rounds plus tôt (avec $DistPred(P) \leq i$), alors la distance entre $P'$ et $P$ est égale à $DistPred(P)$.

**Preuve :** (par récurrence) La seule manière qu'un processeur a de modifier sa variable $DistPred(P)$ est de recopier en incrémentant celle de son prédécesseur (règle $R1$). D'où, si $DistPred(P)$ a une certaine valeur, c'est que le round précédent $DistPred(P^-)$ avait la même valeur *moins* un. A partir de là, le lemme se prouve par récurrence.

1. Si $DistPred(P) = 1$ : alors le round précédent, $DistPred(P^-)$ était à zéro, d'où $P^-$ était privilégié. Dans ce cas, on a $P' = P^-$, d'où la distance entre $P'$ et $P$ est bien de 1.
2. Supposons la propriété vraie pour $DistPred(P) = d - 1$ (avec $d \geq 2$).
3. Si $DistPred(P) = d$ : alors le round précédent, dans la configuration $C_{-1}$, $DistPred(P^-)$ valait $d-1$. On peut alors appliquer l'hypothèse de récurrence et la distance entre $P'$ (le support de $J$ $d-1$ rounds avant $C_{-1}$) et $P^-$ est $d-1$. D'où la distance entre $P$ et $P'$ est de $(d-1)+1$, c'est à dire $d$.

Au final, la propriété est vraie au rang 1 et la supposer au rang $d$ permet de l'établir au rang $d+1$. Elle est donc vraie pour une valeur quelconque de la variable $DistPred$. □

Un exemple illustrant ce lemme est donné figure 8.7. $P$ a sa variable $DistPred$ à 4. 4 rounds plus tôt, $P'$ avait un privilège et la distance entre $P'$ et $P$ est bien de 4.




(a) Etat de P (b) Histoire locale avant le temps T=0

FIG. 8.7 – *Retro Analyse de* $DistPred(P)$

**Lemme 8.7.4**
Soit $C$ une configuration, $P$ un processeur non privilégié et $J$ son privilège successeur. Si $P'$ était le support de $J$ $DistSucc(P)$ rounds plus tôt, alors la distance entre $P'$ et $P$ est égale à $DistSucc(P)$.

**Preuve :** Même principe de preuve que le lemme 8.7.3 □

**Lemme 8.7.5**
Soit $C$ une configuration et $P$ un processeur non privilégié. On suppose que $J$, le privilège prédécesseur de $P$, est un privilège lent. Alors, si $DistPred(P)$ est correcte (et différente de $n$), la distance entre $J$ et $P$ est égale à $\lfloor DistPred(P) \times (1-\frac{1}{4}) \rfloor$ ou à $\lfloor DistPred(P) \times (1-\frac{1}{4}) \rfloor + 1$.

**Preuve :** On pose $d = DistPred(P)$. La variable $DistPred(P)$ étant correcte, on sait que $d$ rounds avant $C$, la distance entre $J$ et $P$ était de $d$. Soit $a$ le nombre de fois où $J$ a avancé en $d$ rounds. $J$ étant un privilège lent, il avance une fois tous les 4 rounds. En $d$ rounds, il a donc avancé $\lfloor d/4 \rfloor$ ou $\lfloor d/4 \rfloor + 1$ fois.

D'autre part, la distance de $J$ à $P$ étant égale à la distance entre $P'$ et $P$ moins le nombre de mouvements que $J$ a fait en $d$ rounds, on a $Dist(J,P) = d - a$, c'est à dire : $Dist(J,P) = d - \lfloor d/4 \rfloor$ ou $Dist(J,P) = d - \lfloor d/4 \rfloor + 1$

Au final, comme $d - \lfloor d/4 \rfloor\} = \lfloor d - d/4 \rfloor$, on a bien $Dist(J,P) = \lfloor d \times 1 - \frac{1}{4} \rfloor$ ou $Dist(J,P) = \lfloor d \times 1 - \frac{1}{4} \rfloor + 1$ □

**Lemme 8.7.6**
Soit $C$ une configuration et $P$ un processeur non privilégié. On suppose que $J$, le privilège successeur de $P$, est un privilège lent. Alors, si $DistSucc(P)$ est correcte, la distance entre $J$ et $P$ est égale à $\lfloor DistSucc(P) \times 1 + \frac{1}{4} \rfloor$ ou à $\lfloor DistSucc(P) \times 1 + \frac{1}{4} \rfloor + 1$.

**Preuve :** Même principe de preuve que le lemme 8.7.5 □

**Lemme 8.7.7**
Soit $\mathcal{C}$ une exécution dont la configuration initiale est légitime. Alors après au plus $n+3$ rounds, le privilège est transmis.

**Preuve :** Soit $J$ le privilège. Si $J$ est un privilège lent, il est transmis au plus tard 4 rounds après le début de l'exécution.

Si $J$ est un privilège rapide, soit $P$ son support. $n+1$ round après son début, l'exécution est dans une configuration $C$ dans laquelle toutes les variables distances sont correctes. $DistPred(P^-)$ vaut donc $n$ (elle pourrait valoir $n+n/4$, mais elle est bornée par $n$). $J$ est donc un privilège actif et est transmis au plus deux transitions après. □

**Lemme 8.7.8**
L'algorithme est correcte.

**Preuve :** Au plus tous les $n+3$ rounds, le privilège est transmis (lemme 8.7.7). Comme il est toujours transmis d'un processeur vers son successeur, tous les processeurs le recevront au cours d'une exécution légitime et la correction est assurée. □

### 8.7.3 Convergence

**Survol de la preuve :** la convergence se prouve en utilisant les lemmes ci-dessus assurant que les variables distances sont nécessairement correctes après un certain temps. Ensuite, à partir d'une exécution dans laquelle les variables distances sont correctes, une fusion au moins à lieu. Le processus itéré assure la convergence.

**Lemme 8.7.9**
Soit $\mathcal{E}$ une exécution. Après $n+1$ rounds, si aucune fusion de privilège n'a eu lieu, l'exécution est dans une configuration où toutes les variables $DistPred$ sont correctes.

**Preuve :** Si aucune fusion n'a lieu, il n'y a pas de disparition de privilège. Soit $C_0$ la configuration initiale de $\mathcal{E}$ et $C_i$ sa $i^{ieme}$ configuration. Montrons que dans $C_{2i+1}$ :

1. Les $i$ successeurs d'un processeur privilégié ont une variable $DistPred$ correcte.
2. Tous les autres processeurs ont dans leur variable $DistPred$ une valeur supérieure ou égale à $i$ :

1. Dans $C_1$ :
   (a) Les variables $DistPred(P)$ de tous les processeurs possédant un privilège sont correctes ;
   (b) Tous les processeurs ont actualisé leurs variables distances. Pour ceux qui n'ont pas de privilège, cette mise à jour consiste à prendre la valeur de leur prédécesseur *plus* un, ce qui est strictement supérieur à zéro.
2. Supposons que dans $C_{2i-1}$,
   (a) tous les successeurs à distance $i$ des processeurs privilégiés aient une variable $DistPred$ correcte et
   (b) tous les autres processeurs aient dans leur variable $DistPred$ une valeur supérieure ou égale à $i$.

3. Considérons $C_{2i+1}$ :
   (a) Si $P$ a un privilège dans $C_{2i-1}$, $DistPred(P^{(i-1)+})$ a une valeur correcte dans $C_{2i-1}$. D'où $DistPred(P^{(i)+})$ a une valeur correcte dans $C_{2i}$ et $DistPred(P^{(i+1)+})$ a une valeur correcte dans $C_{2i+1}$. Or, entre $C_{2i-1}$ et $C_{2i+1}$, $P$ ne peut transmettre son privilège qu'une seule fois.
   D'où, dans $C_{2i+1}$, $P^{(i+1)+}$, c'est à dire le $i^{ieme}$ successeur de $P^+$, a sa variable $DistPred$ correcte.
   (b) Si, dans $C_{2i+1}$, $P$ est à une distance supérieure à $i$ de son privilège prédécesseur, alors, dans $C_{i-1}$, $P^-$ est à une distance supérieure à $i-1$ de son privilège prédécesseur. D'où dans $C_{(i-1)}$, on a $DistPred(P^-) \geq i-1$. D'où, deux rounds plus tard, $P$ ayant pris la valeur de $DistPred(P^-)$ *plus* un, on a $DistPred(P) \geq i$

Au final, la variable $DistPred$ étant bornée par $n$, après $n+1$ rounds, tous les processeurs ayant un privilégié parmi leurs $n$ prédécesseurs ont leur variable $DistPred$ correcte. Les autres ont leur variable $DistPred$ à $n$, ce qui est également correct. □

**Lemme 8.7.10**
Soit $\mathcal{E}$ une exécution. Après $n+1$ rounds, si aucune fusion de privilège n'a eu lieu, l'exécution est dans une configuration où toutes les variables $DistSucc$ sont correctes.

**Preuve :** Cette preuve est grandement similaire à celle du lemme précédent : après $i$ rounds, tous les processeurs ayant un privilégié parmi leur $i$ successeur ont leur variable $DistSucc$ correcte alors que les autres ont dans leur variable $DistSucc$ une valeur supérieure à $i$. D'où, après $n+1$ rounds, tous les processeurs ont leur variable $DistSucc$ correcte. □

**Corollaire 8.7.11**
Soit $\mathcal{E}$ une exécution. Après $2 \times (n+1)$ rounds, si aucune fusion de privilège n'a eu lieu, l'exécution est dans une configuration où toutes les variables distances sont correctes.

**Preuve :** Après $n+1$ rounds, toutes les variables $DistSucc$ sont correctes (lemme 8.7.10) et après $2 \times n + 1$ rounds, toutes les variables $DistPred$ sont également correctes (lemme 8.7.9). D'où toutes les variables distances sont correctes.

□

Un exemple de la mise à jour des variables indices est donné figure 8.8.

**Lemme 8.7.12**
Dans une exécution dont les variables distances sont correctes, la nature (actif ou inactif) des privilèges rapides ne peut changer que si deux privilèges fusionnent.

**Preuve :** Soit $F_2$ un privilège rapide, $J_1$ son prédécesseur et $J_3$ son successeur. Parmi $J_1$ et $J_2$, l'un au moins est un privilège lent. Supposons que ce soit $J_1$.

1. Si $F_2$ est actif : il avance tous les deux rounds. Or $J_1$ est lent, donc la distance entre $F_2$ et $J_1$ augmente tous les quatre rounds. Comme la distance entre $F_2$ et

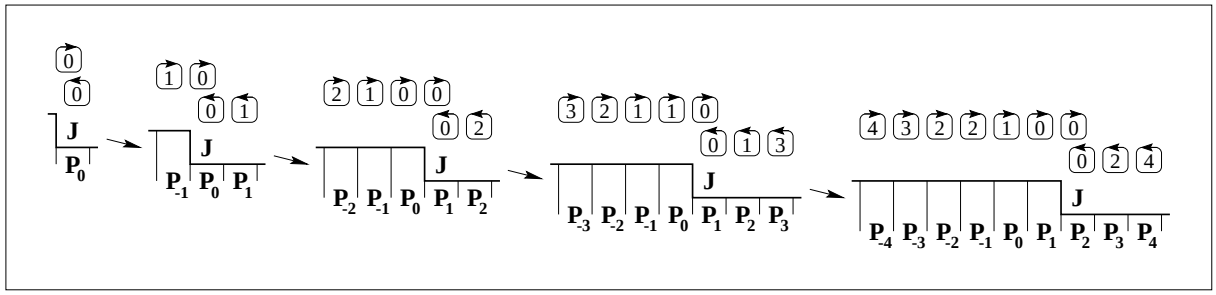



FIG. 8.8 – *Mise à jour des indices*

$J_3$ n'augmente pas (elle reste constante si $J_3$ est un rapide actif, elle diminue dans les autres cas), l'égalité $DistSucc(F_2) \times (1+\frac{1}{4}) \leq DistPred(F_2) \times (1-\frac{1}{4})$ reste vérifiée[5] jusqu'à ce que $J_1$ soit rattrapé ou que $F_2$ rattrape $J_3$.

2. Si $F_2$ est inactif : il n'avance pas. Or $J_1$ avance tous les quatre rounds. Donc la distance entre $F_2$ et $J_1$ diminue tous les quatre rounds. Comme la distance entre $F_2$ et $J_3$ ne diminue pas (elle reste constante si $J_3$ est un rapide inactif, elle augmente dans les autres cas), l'égalité $DistSucc(F_2) \times (1 + \frac{1}{4}) \leq DistPred(F_2) \times (1 - \frac{1}{4})$ reste fausse jusqu'à ce que $F_2$ soit rattrapé ou que $J_3$ rattrape son successeur.

Dans tous les cas, la nature de $F_2$ ne peut pas changer s'il n'y a pas de fusion.

La preuve est symétrique si on suppose que $J_3$ est lent et qu'on ne connaît pas la nature de $J_1$. □

**Lemme 8.7.13**
L'algorithme présenté figure 8.4 converge.

**Preuve :** Soit $\mathcal{E}$ une exécution et soit $2i+1$ (avec $i \geq 1$) le nombre de privilèges présents dans sa configuration initiale. Supposons qu'il n'y ait pas de fusion de privilèges dans $\mathcal{E}$. Après $n+1$ rounds, toutes les variables distances sont correctes (corolaire 8.7.11). La nature des privilèges ne change donc plus (car il n'y a pas de fusion dans $\mathcal{E}$). Si tous les privilèges rapides sont actifs, alors l'algorithme se comporte comme l'algorithme de base et la fusion de deux privilèges est assurée (car l'algorithme de base converge). Si un privilège rapide est inactif, alors sa nature ne pouvant changer avant une fusion, il se fait nécessairement rattraper par son prédécesseur.

Dans tous les cas, une fusion a lieu. Une fusion entraîne la disparition de deux privilèges. D'où, $\mathcal{E}$ contient une configuration dans laquelle $2i-1$ privilèges sont présents. Le même processus itéré $i-1$ fois assure que $\mathcal{E}$ contient une configuration dans laquelle un unique privilège est présent, c'est à dire une configuration légitime.

□

5. On serait tenté de dire "est de plus en plus vérifiée" (car la valeur du terme le plus petit diminue alors que celle du terme le plus grand augmente) ce qui est naturellement faux du point de vue strictement logique mais illustre bien le principe de l'algorithme.

### 8.7.4 Majoration du temps de convergence

**Survol de l'étude :** nous devons montrer que l'algorithme est proportionnel. Plus précisément, étant donné une configuration dont $f$ processeurs sont corrompus, nous allons établir que le temps de convergence est de l'ordre de $O(f^3 Log(f))$ rounds. Pour cela, nous allons distinguer deux cas : lorsque le nombre de fautes est faible ($\sqrt[3]{n}$ processeurs corrompus ou moins), les notions de processeurs corrompus, vrais et faux privilèges peuvent être définies. On montre que l'algorithme converge en $f^3$.

Deuxième cas, lorsqu'une grande partie de l'anneau est corrompue (plus de $\sqrt[3]{n}$ processeurs), alors la convergence au pire est en $nLog(n)$. Au final, dans tous les cas, le temps de convergence est majoré par un polynome en $f$, l'algorithme est bien proportionnel.

**Faible nombre de fautes** dans le cas d'un faible nombre de fautes, on utilise à nouveau les notions de faux privilège et privilège correcteur.

**Définition 8.7.14**
Soit $C$ une configuration illégitime obtenue à partir de la configuration légitime $L$. On distingue dans $C$ deux types de privilèges :

1. $P$ a un privilège *faux* si $Priv(P^-)$ est corrompu.
2. $P$ a un privilège *correcteur* si $Priv(P^-)$ n'est pas corrompu.

**Cas particulier pour moins de $\sqrt[3]{n}$ corruptions contiguës :** dans le cas où tous les processeurs corrompus sont contigus, seul deux processeurs (un corrompu et un correcteur) font leur apparition sur l'anneau. La convergence est alors rapide : initialement et pendant quelques rounds, un faux privilège rapide peut être actif à cause des informations erronées qu'il reçoit sur les distances. Mais assez rapidement, il devient inactif et l'exécution converge.

**Lemme 8.7.15**
Soit $\mathcal{E} = (C_0, C_1, ...)$ une exécution et $f$ le nombre initial de fautes. Alors, si $J_1$ et $J_2$ n'ont pas fusionné avant la configuration $C_{2f}$, $C_{2f}$ vérifie que :

1. Les voisins du faux privilège, les voisins du privilège correcteur et les $f$ successeurs du faux privilège ont des variables distances correctes.
2. Le nombre de corrompus n'exède pas $2f$.

**Preuve :** Dans $C_0$, soit $P_2$ le support de $J_2$. Dans $C_{2f}$, les $2f$ successeurs de $P_2$ ont mis à jour leur variable $DistPred$. D'un autre côté, comme ils ne sont pas corrompus, leur variable $DistSucc$ est correcte. Dans le même temps, $J_2$ a au plus avancé $f$ fois. D'où, dans $C_{2f}$, les $f$ successeurs de $J_2$ ont leurs variables distances correctes.

De la même manière, si $P_1$ est le support de $J_1$ dans $C_0$, les $2f$ successeurs de $J_1$ et les $2f$ prédécesseurs de $J_2$ ont actualisé leurs variables distances. Tous les processeurs entre $J_1$ et $J_2$ ont donc des variables distances correctes. Le pré-

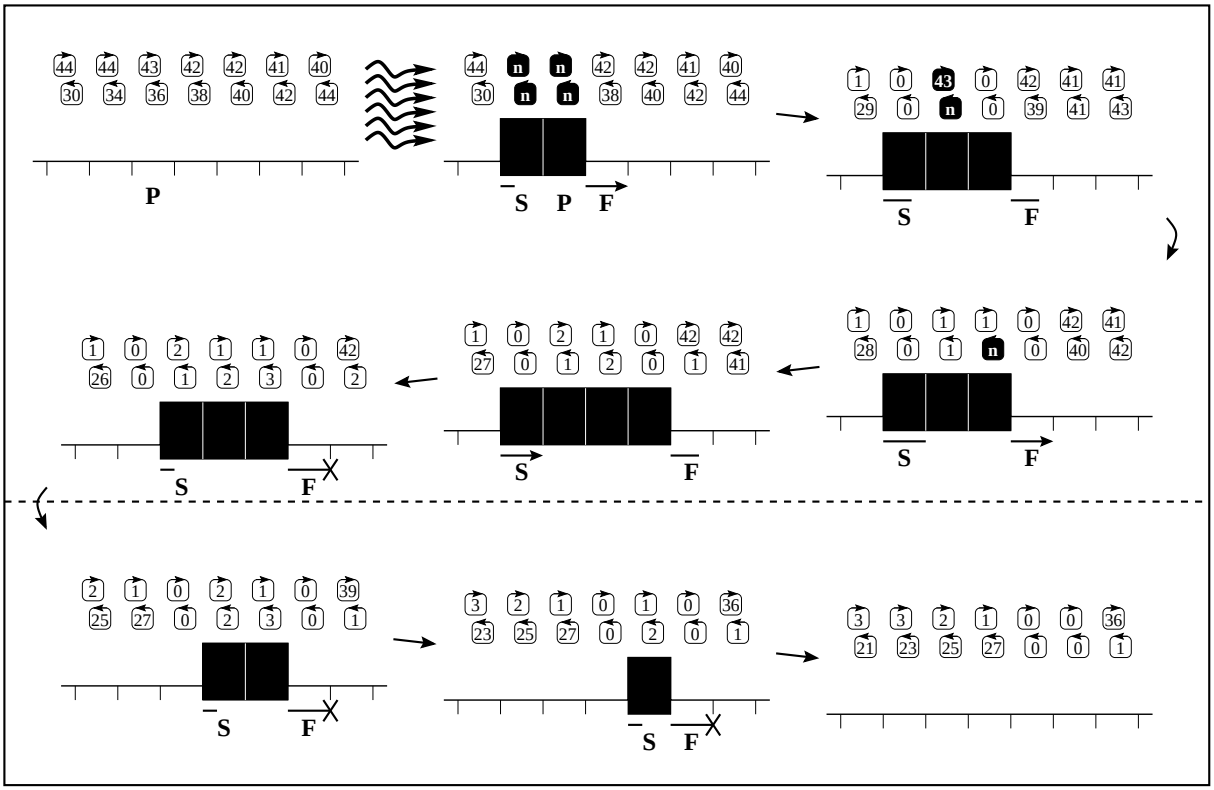



FIG. 8.9 – *Correction des indices*

décesseur de $J_1$ n'étant pas corrompu,les voisins de $J_1$ et $J_2$ ont effectivement des variables distances correctes.

D'un autre côté, en $2f$ rounds , la distance entre $J_1$ et $J_2$ peut avoir augmenté au plus de $f$ (si $J_2$ est rapide actif et que $J_1$ est rapide inactif). D'où le nombre de corrompus ne peut excéder $f + f$. □

La figure 8.9 (au dessus des pointillés) illustre ce lemme. Initialement, $P$ et $P^-$ sont corrompus. $F$ est un faux privilège rapide actif de par les fausses variables distances qui lui parviennent. Néanmoins, assez rapidement, les variables distances se corrigent et $F$ devient inactif. Entre la configuration initiale et le moment où $F$ reçoit des informations justes (configuration $C_5$), le nombre de corrompus a été jusqu'à doubler (mais il n'aurait pas pu augmenter plus ; en effet, le cas considéré est le pire possible : $F$ est sur le point d'être transmis, $S$ doit attendre 4 rounds avant d'être transmis et tous les indices de $P$ et $P^-$ sont a $n$).

**Lemme 8.7.16**
Si les $f$ processeurs corrompus sont contigus, l'algorithme converge en $12f$ rounds.

**Preuve :** Seuls deux processeurs font leur apparition sur l'anneau. Soit $J_2$ le faux privilège et $J_1$ le privilège correcteur. Après $2f$ rounds, dans la configuration $C_{2f}$, $J_1$ et $J_2$ ont des voisins ayant des variables distances correctes. Ils peuvent donc connaître les distances les séparant de leurs privilèges voisins. A partir de là,

plusieurs cas de figure sont possibles :

1. Si $J_1$ est un privilège rapide : $J_1$ peut être actif ou inactif.
   (a) Si $J_1$ est inactif ($DistPred(J_1^-) \times 0{,}75 < DistPred(J_1^+) \times 1{,}25$) : cela signifie que $J_1$ est plus proche de son privilège prédécesseur $J_0$ que de son privilège successeur (sauf si $J_0$ est un privilège rapide, auquel cas $J_1$ est encore plus proche de $J_0$ qu'il ne le croit). Dans tous les cas, $J_1$ est inactif et après au plus $4 \times Dist(J_0,J_1)$, $J_0$ et $J_1$ fusionnent.
   La distance $Dist(J_0,J_1)$ est inférieure à la distance $Dist(J_1,J_2)$ (sinon, $J_1$ serait actif) elle même inférieure à $2f$. D'où $J_1$ et $J_2$ fusionnent après au plus $8f$ rounds.
   (b) Si $J_1$ est actif et que $RejointD(J_1)$ est plus petit que $RejointD(J_2)$ : alors la distance entre $J_1$ et $J_2$ étant au plus de $2f$, $J_1$ rejoint $J_2$ en $4 \times 2f$ rounds.
   (c) Si $J_1$ est actif et que $RejointD(J_1)$ est plus grand que $RejointD(J_2)$ : alors après $RejointD(J_2)/4$ rounds, $J_2$ rejoint $D$ et devient un privilège rapide. A ce moment, la distance entre $J_1$ et $J_2$ n'est plus que de $2f + RejointD(J_2)/4 - RejointD(J_2)/4$.
   Si $J_2$ devient actif, c'est que son privilège successeur est plus proche de $J_2$ que ne l'est $J_1$ et ils fusionnent en moins de $4 \times (2f - RejointD(J_2)/4)$ rounds. Si $J_2$ devient inactif, $J_1$ le rattrape en moins de $4 \times (2f - RejointD(J_2)/4)$ rounds. Dans tous les cas, la fusion de $J_1$ avec un autre privilège a lieu moins de $RejointD(J_2)/4 + 4 \times (2f - RejointD(J_2)/4) \leq 8f$ rounds

   Dans tous les cas, une fusion intervient au plus $8f$ rounds après $C_{2f}$, c'est à dire $10f$ rounds après le début de l'exécution.
2. Si $J_1$ est un privilège lent : $J_2$ peut être actif ou inactif.
   (a) Si $J_2$ est inactif : la distance entre $J_1$ et $J_2$ est au plus de $2f$, d'où $4 \times 2f$ rounds plus tard, $J_1$ rattrape $J_2$.
   (b) Si $J_2$ est actif : $J_2$ est plus proche de son privilège successeur $J_3$ que de $J_1$. La distance entre $J_2$ et $J_3$ est donc inférieure à $2f \times 1{,}5/1{,}25 = 2{,}4f$ et donc, au plus tard $4 \times 2{,}4f$ rounds plus tard, la fusion a lieu.

Dans tous les cas, une fusion intervient au plus $10f$ rounds après $C_{2f}$, c'est à dire $12f$ rounds après le début de l'exécution. □

La figure 8.9 illustre ce lemme. Après une phase de correction des indices et ou le nombre de corrompu peut augmenter (jusqu'à doubler), le faux privilège rapide $F$ devient inactif et se fait rattraper par le privilège lent $S$, le tout en 16 rounds (ce qui est inférieur à $(12 \times 2)$.

**Cas général pour moins de $\sqrt[3]{n}$ corruptions non contiguës** Lorsque les processeurs corrompus sont contigus, la convergence est rapide (de l'ordre de $f$). De même, si plusieurs groupes de processeurs corrompus sont spacialement éloignés les uns des autres, on a l'assurance qu'une fois les variables distances corrigées, tous les faux privilèges rapides vont devenir inactifs et se faire rattrapper. Mais cette certitude disparaît

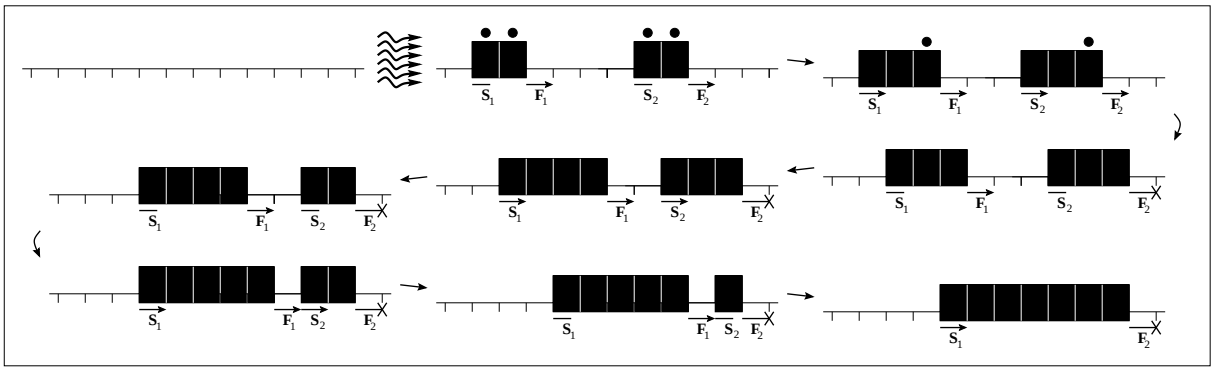



FIG. 8.10 – *Un faux privilège rapide... actif*

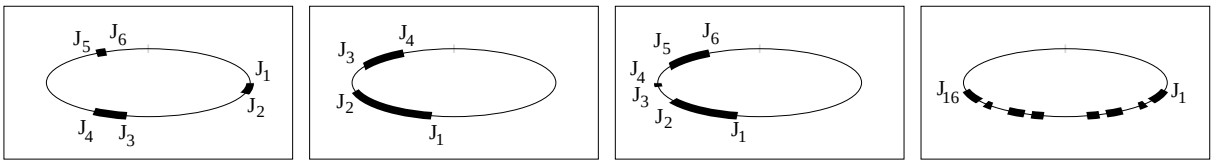



FIG. 8.11 – *Exemples de blocs*

en cas de groupe de corrompus proches les uns des autres. Plus particulièrement, un faux privilège rapide peut initialement avancer suite à une corruption des variables distances, puis se trouver plus proche de son privilège successeur que de son privilège prédécesseur. Il continue à être actif augmentant ainsi le nombre de corrompus présents sur l'anneau. La figure 8.10 illustre ce problème. Chaque flèche représente deux transitions. Le détail des variables indices n'est pas donné mais un point noir représente les indices qui peuvent encore avoir une fausse valeur suite à la corruption initiale.

Il convient donc de distinguer deux types différents de groupe de processeurs corrompus : ceux qui sont loin les uns des autres (et qui ne vont pas interférer entre eux pendant la convergence) et ceux qui au contraire sont proches, comme $S_1$, $F_1$, $S_2$ et $F_2$ sur la figure 8.10. Quand plusieurs groupes sont proches, on dit qu'ils appartionnent au même bloc de processeurs corrompus, ou plus simplement au même bloc.

**Définition 8.7.17**
On dit d'une une suite de privilèges ($J_1$,$J_2$,...,$J_l$) que c'est un *bloc de privilèges* si au cours de l'exécution, $J_1$ fusionne avec $J_l$ et s'il n'existe pas de prédécesseur de $J_1$ fusionnant avec un successeur de $J_l$

Des exemples de blocs sont donnés figure 8.11. La notion de bloc ne s'appliquant qu'à des anneaux de grande taille[6], nous changeons momentanément la représentation

6. Nous traitons ici le cas où le nombre de fautes est petit devant la taille de l'anneau ($f \leq \sqrt[3]{n}$). Si l'anneau est lui-même de petite taille, il ne peut y avoir que très peu de fautes, ce qui limite les possibilités de séparer les processeurs corrompus en groupe distincts. Par exemple, sur un anneau de taille 19 ou 23 (tailles des anneaux généralement utilisés dans nos figures jusqu'à présent), au plus 2

graphique des configurations. Elles sont maintenant symbolisées par des ellipses dont chaque point est un processeur (leur nombre étant trés grand, ils ne sont pas dessinés individuellement). Les processeurs corrompus sont représentés par un trait. Ainsi,  est une portion d'anneau non corrompue,  est un petit groupe de processeurs corrompus alors que  est un groupe de processeurs corrompus un peu plus grand.

Dans la configuration $C_1$, les trois groupes de processeurs forment trois blocs distincts. En effet, la distance entre $J_1$ et $J_2$ est petite (relativement à celle entre $J_1$ et $J_6$ ou à celle entre $J_2$ et $J_3$). $J_1$ va donc fusionner avec $J_2$ avant d'être rattrappé. De même, $J_3$ va fusionner avec $J_4$ et $J_5$ va fusionner avec $J_6$.

Dans $C_2$, $J_2$ va rattrapper $J_3$ et seulement ensuite $J_1$ va rattrapper $J_4$. Les deux groupes de corrompus ne forment donc qu'un seul bloc.

Dans $C_3$, $J_3$ va fusionner avec $J_4$ (et donc le groupe de processeurs corrompus entre $J_3$ et $J_4$ va disparaître) mais $J_2$ va ensuite rattrapper $J_5$. Au final, $J_1$ va fusionner avec $J_6$. Tous les corrompus de $C_3$ (y compris ceux situés entre $J_3$ et $J_4$) sont donc dans le même bloc.

La configuration $C_4$ illustre le proverbe populaire cité en tête de chapitre : de petits groupes de corrompus séparés par de de petits groupes de non corrompus peuvent fusionner, constituant ainsi des groupes de corrompus un peu plus gros. Ces derniers peuvent à leur tour fusionner entre eux, augmentant ainsi le nombre global de corrompus. Ainsi, tous les groupes de corrompus de la configuration $C_4$ sont dans le même bloc.

**Notation :** Dans tout ce qui suit, nous utilisons des notations un peu particulières. Dans une suite de privilèges, un privilège sur deux est un corrompu, l'autre est correcteur. Les deux privilèges créés par le même ensemble de fautes sont notés avec le même indice. Ainsi, une suite de $2l$ privilèges est notée $(J_1,J'_1,J_2,J'_2,...,J_l,J'_l)$ (c'est à dire $(S_1,F_1,S_2,F_2,...,S_l,F_l)$ si le premier processeur de la suite est un processeurs lent[7] et $(F_1,S_1,F_2,S_2,...,F_l,S_l)$ dans le cas inverse).

De plus, au round $t$, $f_i(t)$ désigne la distance séparant le privilège correcteur $J_i$ et son privilège successeur $J'_i$ (par exemple, $f_3(t)$ est la distance entre $S_3$ et $F_3$) et $h_i(t)$ désigne la distance séparant $J'_i$ de son privilège successeur $J_{i+1}$ (par exemple, $h_3(t)$ est la distance entre $F_3$ et $S_4$).

**Définition 8.7.18**
La taille du bloc $\mathcal{B} = (J_1,J'_1,J_2,J'_2,...,J_l,J'_l)$ à l'instant $t$ est le nombre de processeurs corrompus dans $\mathcal{B}$.

$$Taille_{\mathcal{B}}(t) = \sum_{j=1}^{l} f_i(t)$$

On remarque que, pour une exécution donnée, la somme de tous les $f_i(0)$ est le nombre initial de corrompus.

processeurs pourraient être corrompus (car $3 > \sqrt[3]{23}$). Cela ne permettrait pas d'illustrer le concept de bloc.

7. Jusqu'à présent, nous notions $(S_1,F_2,S_3,F_4,...,S_{2l-1},F_{2l})$ ce genre de suite.

**Lemme 8.7.19**
Soit un bloc de $2l$ privilèges $(S_1,F_1,S_2,...,S_l,F_l)$ tel que le processeur distingué ne soit pas entre deux des privilèges et tel que $F_l$ ne franchisse pas $D$ au cours de la stabilisation. Si $F_i$ ou un de ses prédécesseurs rattrappe $S_i$ (ou un de ses successeurs), alors $h_i(0) \leq 2f_i(0) + \sum_{j=1}^{i-1}(f_j(0) + h_j(0))$.

**Preuve :** Soit $F_\alpha$ le prédécesseur de $F_i$ (ou éventuellement $F_i$ lui même) qui va rattrapper $S_{i+1}$ ou un de ses successeurs.

1. Si $h_i(0) \leq 2f_i(0)$, le résultat est immédiat.
2. Sinon, après $2f_i(0)$ rounds, $F_i$ devient un processeur inactif. Soit $P_i$ son support. Quand $F_\alpha$ arrive en $P_i$, la distance (noté $f_\alpha(t)$) qui le sépare de son privilège prédécesseur est majorée par [la distance initiale entre $S_1$ et $F_i$] plus [le nombre de fois où $F_i$ a avancé] moins [le nombre de fois où $S_1$ a avancé], c'est à dire $f_\alpha(t) \leq [f_i(0) + \sum_{j=1}^{i-1}(f_j(0) + h_i(0))] + [f_i(0)] - [t/4]$. Parallèlement, la distance (notée $h_\alpha(t)$) qui sépare $F_\alpha$ de son privilège successeur est minorée par [la distance initiale entre $F_i$ et $S_{i+1}$] moins [le nombre de fois où $F_i$ a avancé] plus [le nombre de fois où $S_{i+1}$ a avancé], c'est à dire $h_\alpha(t) \geq h_i(0) - f_i(0) + t/4$. Comme au cours de l'exécution $F_\alpha$ rattrape un de ses successeurs, il doit redevenir actif. Il doit donc exister un round $t$ tel que $h_\alpha(t) \leq f_\alpha(t)$, c'est à dire :

$$\begin{array}{rcl} h_\alpha & \leq & f_\alpha \\ h_i(0) - f_i(0) + t/4 & \leq & f_i(0) + \sum_{j=1}^{i-1}(f_j(0) + h_j(0)) + f_i(0) - t/4 \\ h_i(0) & \leq & 3f_i(0) + \sum_{j=1}^{i-1}(f_j(0) + h_j(0)) - t/2 \\ h_i(0) & \leq & 3f_i(0) + \sum_{j=1}^{i-1}(f_j(0) + h_j(0)) - 2f_i(0)/2 \\ h_i(0) & \leq & 2f_i(0) + \sum_{j=1}^{i-1}(f_j(0) + h_j(0)) \end{array}$$

.

Au final, on a bien $h_i(0) \leq 2f_i(0) + \sum_{j=1}^{i-1}(f_j(0) + h_j(0))$. □

**Lemme 8.7.20**
Soit un bloc de $2l$ privilèges $(F_1,S_1,F_2,...,F_l,S_l)$ tel que le processeur distingué ne soit pas entre deux des privilèges et tel que $F_l$ ne franchisse pas $D$ au cours de la stabilisation. Si $S_i$ ou un de ses prédécesseurs rattrappe $F_i$ (ou un de ses successeurs), alors $h_i(0) \leq f_{i+1}(0) + \sum_{j=i+1}^{l-1}(f_j(0) + h_{j+1}(0))$.

La preuve de ce lemme est symétrique à celle du lemme 8.7.19.

**Lemme 8.7.21**
Soit un bloc de $2l$ privilèges $(S_1,F_1,S_2,...,S_l,F_l)$ tel que le processeur distingué ne soit pas entre deux des privilèges et tel que $F_l$ ne franchisse pas $D$ au cours de la stabilisation. Alors, si $l = 2^i$, on a $Dist(S_1,F_l) \leq 4^i Taille_{\mathcal{B}}(0)$.

**Preuve :** Par récurrence sur $i$ :

1. Si $i = 1$ : la suite est composée de 4 privilèges $S_1$, $F_1$, $S_2$ et $F_2$. Comme $S_1$ fusionne avec $F_2$, on a $h_1(0) \leq 2f_1(0)$ (lemme 8.7.19).

Comme $f_1(0)$ est plus petit que $Taille_{\mathcal{B}}(0)$, on a bien $Dist(S_1,F_2) = f_1(0) + h_1(0) + f_2(0) \leq 3f_1(0) + f_2(0) \leq 4Taille_{\mathcal{B}}(0)$

2. Supposons que pour $i$, la propriété soit vraie.
3. Montrons là pour $i+1$ : on a $Dist(S_1,F_{2^{i+1}}) = Dist(S_1,F_{2^i}) + h_{2^i} + Dist(S_{2^i+1},F_{2^{i+1}})$. Or $S_1$ fusionne avec $F_l$, d'ou $h_{2^i} \leq 2f_{2^i}(0) + \sum_{j=1}^{2^i-1}(f_j(0)+h_j(0)) \leq 2Dist(S_1,F_{2^i})$. Par hypothèse de récurrence, $Dist(S_1,F_{2^i})$ et $Dist(S_{2^i+1},F_{2^{i+1}})$ sont inférieures à $4^iTaille_{\mathcal{B}}(0)$. On a donc

$$\begin{array}{rcccccc} Dist(S_1,F_{2^i}) & = & Dist(S_1,F_{2^i}) & + & h_{2^i} & + & Dist(S_{2^i+1},F_{2^{i+1}}) \\ & \leq & 4^iTaille_{\mathcal{B}}(0) & + & 2 \times 4^iTaille_{\mathcal{B}}(0) & + & 4^iTaille_{\mathcal{B}}(0) \\ & \leq & & & 4^{i+1}Taille_{\mathcal{B}}(0) & & \end{array}$$

□

**Lemme 8.7.22**
Soit un bloc de $2l$ privilèges $(J_1,J'_1,J_2,...,J_l,J'_l)$.
Alors, si $l = 2^i$, on a $Dist(J_1,J'_l) \leq 4^iTaille_{\mathcal{B}}(0)$.

**Preuve :** Que le bloc de privilège soit de la forme

1. $(S_1,F_1,S_2,...,S_l,F_l)$ avec la propriété que $F_l$ franchit le distingué,
2. $(S_1,F_1,S_2,...,F_{j-1},S_j,S'_j,F_{j+1},...,F'_l,S'_l)$ avec le distingué situé entre $S_j$ et $S'_j$,
3. $(F_1,S_1,F_2,...,F_l,S_l)$ avec la propriété que $S_l$ ne franchit pas le distingué,
4. $(F_1,S_1,F_2,...,F_l,S_l)$ avec la propriété que $S_l$ franchit le distingué ou
5. $(F_1,S_1,F_2,...,S_{j-1},F_j,F'_j,S_{j+1},...,S'_l,F'_l)$ avec le distingué situé entre $F_j$ et $F'_j$,

la preuve est la même que celle du lemme 8.7.21.

□

**Lemme 8.7.23**
Soit un bloc de $2l$ privilèges $J_1$, ... $J'_l$. Alors pour tout $t$, on a $Dist(J_1,J'_l) \leq 4 \times Taille_{\mathcal{B}}(0)^3$.

**Preuve :** Soit $f'$ la taille initiale du bloc $\mathcal{B}$ ($f' = Taille_{\mathcal{B}}(0)$). Soit $i$ le plus petit entier tel que $2^i > l$. On a $Log_2(l) < i \leq 1 + Log_2(l)$. La taille de $\mathcal{B}$ est inférieure à la distance séparant $J_1$ de $J'_l$. Or $Dist(S_1,F_l) \leq 4^if'$ (lemme 8.7.21 ou 8.7.22). D'où :

$$\begin{array}{rcl} Dist(J_1,J'_l) & \leq & 4^if' \\ & \leq & e^{i \times Log(4)}f' \\ & \leq & e^{(1+Log_2(l)) \times Log(4)}f' \\ & \leq & e^{Log(4)}e^{Log(l)/Log(2) \times Log(4)}f' \\ & \leq & 4e^{Log(l) \times 2}f' \\ & \leq & 4l^2f' \\ & \leq & 4f'^3 \end{array}$$

□

**Corollaire 8.7.24**
Au cours d'une exécution, La taille maximum d'un bloc $\mathcal{B}$ est de $4 \times Taille_{\mathcal{B}}(0)^3$

**Preuve :** La taille maximum d'un bloc est plus petite que la distance entre le premier et de dernier processeur du bloc. D'où $Taille_{\mathcal{B}}(t) \leq Dist(J_1,J'_l) \leq 4 \times Taille_{\mathcal{B}}(0)^3$. □

**Lemme 8.7.25**
Soit un bloc de $2l$ privilèges $(J_1,...,J'_l)$ tel que le processeur distingué ne soit pas entre deux des privilèges et tel que $J_l$ ne franchisse pas $D$ au cours de la stabilisation.
Alors après au plus $18 \times Taille_{\mathcal{B}}(0)^3$ rounds, $J_1$ fusionne avec $J'_l$.

**Preuve :** La taille maximum du bloc est de $4Taille_{\mathcal{B}}(0)^3$ (corollaire 8.7.24).

Si $J_1$ est un privilège rapide, en $4 \times 4 \times Taille_{\mathcal{B}}(0)^3$ rounds, il avance $8Taille_{\mathcal{B}}(0)^3$ fois alors que $S_l$ avance $4Taille_{\mathcal{B}}(0)^3$ fois. D'où, $F_1$ et $S_l$ fusionnent.

Si $J_1$ est un privilège lent, $J'_l$ est un privilège rapide. Après au pire $2f'$ rounds, $J'_l$ devient inactif puis $4 \times 4 \times Taille_{\mathcal{B}}(0)^3$ round plus tard, $S_1$ fusionne avec $F_l$.

Dans tous les cas, après $18 \times Taille_{\mathcal{B}}(0)^3$ rounds, $J_1$ fusionne avec $J'_l$. □

**Lemme 8.7.26**
Soit un bloc de $2l$ privilèges $(J_1,...,J'_l)$. Alors après au plus $34 \times Taille_{\mathcal{B}}(0)^3$ rounds, $J_1$ fusionne avec $J'_l$.

**Preuve :** Si $D$ n'est pas entre deux privilèges de $\mathcal{B}$ et si $J'_l$ ne franchit pas $D$ au cours de l'exécution, la convergence a lieu en $18 \times Taille_{\mathcal{B}}(0)^3$ rounds (lemme 8.7.25).

Sinon, après au plus $4 \times Dist(J_1,D)$, $J_1$ franchit $D$, c'est à dire après au plus $4 \times 4 \times Taille_{\mathcal{B}}(0)^3$ (lemme 8.7.23.

Ensuite, après au plus $18 \times f'^3$ rounds, $S_1$ fusionne avec $J'_l$ (lemme 8.7.25).

Au final, la convergence a lieu moins de $34 \times Taille_{\mathcal{B}}(0)^3$ rounds après le début de l'exécution. □

**Lemme 8.7.27**
Si les $f$ processeurs corrompus forment un unique bloc de $2l$ privilèges, l'algorithme converge en $O(f^3)$ rounds.

**Preuve :** Après au plus $34 \times f^3$ rounds, le premier privilège du bloc fusionne avec le dernier. Comme tous les corrompus sont dans un seul bloc, cela assure la disparition de tous les corrompus et l'algorithme atteint une configuration légitime en $O(f^3)$ rounds. □

**Lemme 8.7.28**
Soit $\mathcal{E}$ une exécution dont la configuration initiale comporte $f$ processeurs corrompus. Alors $\mathcal{E}$ convergence en $O(f^3)$ rounds.

**Preuve :** Etant donné la configuration initiale d'une exécution, il existe une unique manière de regroupper les processeurs corrompus sous forme de blocs dis-

tincts. Le temps de convergence est alors le pire des temps de convergence de tous les blocs, c'est à dire au pire $34f^3$ rounds. □

**Nombre de fautes supérieures à $\sqrt[3]{n}$ :** nous allons montrer que l'algorithme converge en $O(nLog(n))$ rounds.

**Lemme 8.7.29**
Si la configuration initiale ne comporte que 3 privilèges, l'algorithme converge en $6n$ rounds.

**Preuve :** Après au plus $n$ transitions, tous les indices sont corrects. On suppose que sur les trois privilèges, deux sont lents (si ce n'est pas le cas, après au plus $2n$ transitions, un privilège rapide franchit $D$ et l'exécution atteint alors une configuration $C_0$ dans laquelle deux des trois privilèges sont lents). Soit $S_1$, $F_2$ et $S_3$ les trois privilèges dans $C_0$.

1. Si $Dist_{C_0}(S_1,F_2) < Dist_{C_0}(F_2,S_3)$ : alors $F_2$ est inactif et est rejoint par $S_1$ en $4 \times Dist_{C_0}(S_1,F_2) \leq 2n$ rounds (voir figure 8.12.(a)).
2. Si $Dist_{C_0}(S_1,F_2) \geq Dist_{C_0}(F_2,S_3)$ : alors $F_2$ est actif. Deux cas peuvent se présenter :
   (a) Si $RejointD_{C_0}(F_1) \leq RejointD_{C_0}(S_3)$ : alors $F_2$ va rattrapper $S_3$ avant que celui-ci ne franchisse $D$, c'est à dire en moins de $2 \times RejointD_{C_0}(F_3) \leq 2 \times (n/2)$ rounds (voir figure 8.12.(b)).
   (b) Si $RejointD_{C_0}(F_1) > RejointD_{C_0}(S_3)$ : alors, dans la configuration $C_1$, $S_3$ franchit $D$ et devient $F_3$. Là encore, deux cas sont possibles :
      i. Si $Dist_{C_1}(F_3,S_1) > Dist_{C_1}(F_2,F_3)$ : alors $F_3$ est inactif et $F_2$ le rattappe en $2 \times RejointD_{C_0}(F_2)$ rounds (voir figure 8.12.(c)).
      ii. Si $Dist_{C_1}(F_3,S_1) \leq Dist_{C_1}(F_2,F_3)$ : alors $F_3$ est actif. Comme $Dist_{C_1}(F_3,S_1) \leq Dist_{C_1}(F_2,F_3) \leq n$, on a $Dist_{C_1}(F_3,S_1) \leq n/2$ et $F_3$ va rattrapper $S_1$ en au plus $2n$ rounds (voir figure 8.12.(d)).

Dans tous les cas, après au plus $n + 2n + Max\{2n,n,2n,3n\} = 6n$ rounds, une fusion a lieu et l'algorithme converge. □

**Lemme 8.7.30**
Soit $\mathcal{E}^{2i+1}$ une exécution dont la configuration initiale comporte $2i+1$ privilèges. Alors il existe $\mathcal{E}^{2i-1}$, une exécution dont la configuration initiale comporte $2i+1$ privilèges tel que $\mathcal{E}^{2i+1}$ soit C-équivalente à $\mathcal{E}^{2i-1} + \frac{10n}{2i+1}$.

**Preuve :** Soit $(J_1,...,J_{2i+1})$ la suite des privilèges de l'anneau. Soit $j$ tel que $h_j(t_0)$ soit la plus petite distance entre deux privilèges $J_j$ et $J_{j+1}$ (avec $J \in [1..2i+1]$ *modulo* $(2i+1)$). Au temps $t_1$, après $2h_j(0)$ rounds, $J_{j+1}$ connait la distance à son prédécesseur et $J_j$ connait la distance à son successeur. Cette distance est d'au plus $2h_j(0)$ processeurs. D'où au plus $8h_j(0)$ rounds plus tard, une fusion de privilège à lieu (il est possible qu'elle n'ait pas lieu entre $J_j$ et $J_{j+1}$, par exemple si $J_{j+1}$ est maintenant plus proche de son successeur que de $J_j$. Mais elle a alors lieu entre $J_{j+1}$ et $J_{j+2}$ et cela dans un temps plus court que $8h_j(0)$ puisque $Dist(J_{j+1},J_{j+2})$

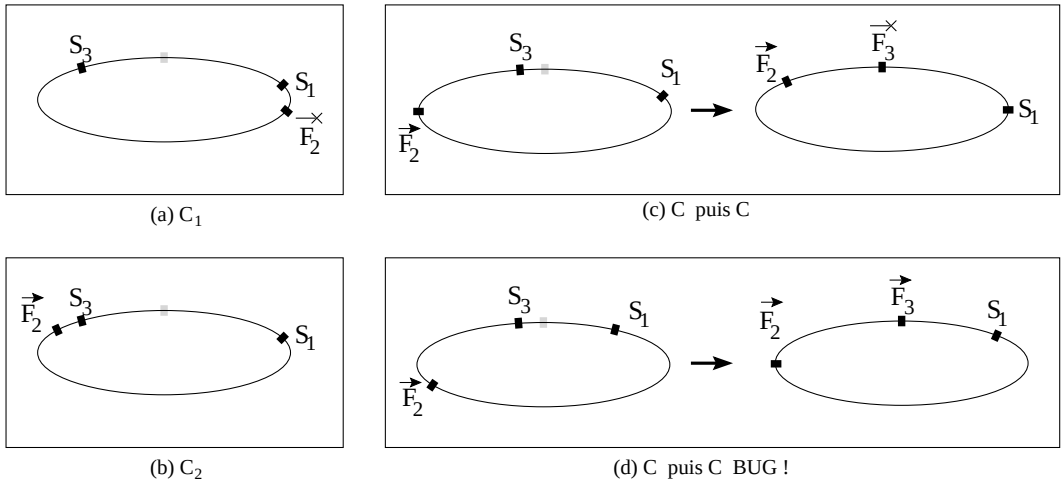



FIG. 8.12 – *Convergence avec trois privilèges*

est plus petit que $2h_j(0)$). L'exécution atteint alors une configuration dans laquelle $2i-1$ processeur sont corrompus. Pour se faire, elle a mis $10h_j(0)$ rounds. Comme il y a $2i+1$ privilèges sur l'anneau et que $h_j(0)$ est la plus petite distance entre deux privilèges, on a $h_j(0) \leq \frac{n}{2i+1}$. Donc $\mathcal{E}^{2i+1}$ est C-équivalente a $\mathcal{E}^{2i-1} + \frac{10n}{2i+1}$. □

**Lemme 8.7.31**
L'algorithme converge en $O(nLog(n))$ rounds.

**Preuve :** Une exécution dont la configuration initiale comporte 3 privilèges converge en $6n$ rounds. Une exécution dont la configuration initiale comporte $2i+1$ privilèges converge en $\frac{10n}{2i+1} + \frac{10n}{2i-1} + \frac{10n}{2i-3} + ...\frac{10n}{5} + 6n$ rounds. D'où, une exécution dont la configuration initiale comporte $n$ privilèges converge en

$$\begin{aligned} 6n + 10n \sum_{j=2}^{\frac{n-1}{2}} (\frac{1}{2j+1}) &= 6n + 5n \sum_{j=2}^{\frac{n-1}{2}} \frac{1}{j+\frac{1}{2}} \\ &\leq 6n + 5n \int_2^{1+\frac{n-1}{2}} \frac{1}{j+\frac{1}{2}} dj \\ &\leq 6n + 5n[Log(j+\tfrac{1}{2})]_2^{\frac{n+1}{2}} \\ &\leq 6n + 5n(Log(\tfrac{n+2}{2}) - Log(\tfrac{5}{2})) \end{aligned}$$

D'où, l'algorithme converge en $O(nLog(n))$ rounds. □

### Complexité globale

**Lemme 8.7.32**
L'algorithme est proportionnel. Si $f$ est le nombre intial de fautes, son temps de convergence est en $O(f^3)$ rounds si $f$ est inférieur à $\sqrt[3]{n}$ et en $O(nLog(n))$ sinon.

**Preuve :** Soit $\mathcal{E}_f$ une exécution dont la configuration initiale $C_0$ comporte $f$ fautes. Si $f$ est inférieur à $\sqrt[3]{n}$, l'algorithme converge en $O(f^3)$ transtions, c'est à dire qu'il existe une constante $c_1$ vérifiant $TempsConv(\mathcal{E}_f) \leq c_1 f^3$. Si $f$ est

supérieur à $\sqrt[3]{n}$, il converge en $O(nLog(n))$ transitions, c'est à dire qu'il existe une constante $c_2$ vérifiant $TempsConv(\mathcal{E}_f) \leq c_2 nLog(n)$. Considérons le polynome $P(f) = (c_1 + 3c_2)f^4$.

1. Si $1 \leq f \leq \sqrt[3]{n}$ : alors $c_1 f^3$ est plus petit que $(c_1 + 3c_2)f^4$ et le temps de convergence de $\mathcal{E}_f$ est majoré par $P(f)$.
2. Si $\sqrt[3]{n} < f \leq n$ : alors $n$ est plus petit que $f^3$ et :

$$\begin{array}{rcl} c_2 nLog(n) & \leq & c_2 f^3 Log(f^3) \\ & \leq & 3c_2 f^3 Log(f) \\ & \leq & 3c_2 f^4 \\ & \leq & P(f) \end{array}$$

Le temps de convergence de $\mathcal{E}_f$ est donc bien majoré par $P(f)$.

D'où, dans tous les cas, l'algorithme a un temps de convergence de l'ordre de $O(f^4)$ rounds. Il vérifie donc la caractéristique générale des algorithmes proportionnels.

□

# Bibliographie

[1] Y. Afek, B. Awerbuch, S. A. Plotkin and M. Saks. Local Management of a Global Resource in a Communication Network. In *Proceedings of the 28th FOCS*, October 1987

[AB93] Y. Afek et G.M. Brown. *Self-stabilization over unreliable communication media.* In *Distributed Computing*, Vol. 7, pages 27-34, 1993.

[AB97] Y. Afek and A. Bremler . Self-Stabilizing Unidirectional Network Algorithms by Power-Supply. *Chicago Journal of Theoretical Computer Science*, to appear.

[ABBR91] A. Arnold, J. Beauquier, B. Bérard, B. Rozoy. *Programmes parallèles : Modèles et validation.* Ed. Armand Colin, 1991.

[AD97] Y. Afek and S. Dolev. Local stabilizer. In *Proceedings of the 5th Israel Symposium on Theory of Computing and Systems*, June 1997.

[ADHK97] U. Abraham, S Dolev, T Herman et I Koll. "Self-stabilizing L-exclusion". Dans *WSS97*, pp 48-63, 1997.

[AG91] Yehuda Afek et Eli Gafni. "Time and message bounds for election in synchronous and asynchronous complete networks". Dans *SIAM Journal on Computing*, 20(2):376-394, avril 1991.

[AK93] Sudhanshu Aggarwal et Shay Kutten. "Time optimal Self-Stabilizing Spanning Tree Algorithms". 1993.

[2] B. Awerbuch, Shay Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time optimal self-stabilizing synchronization. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing, San Diego, California*, pages 652–661, May 1993.

[3] Y. Afek, Shay Kutten, and M. Yung. Memory-efficient self-stabilization on general networks. In *Proc. 4th Workshop on Distributed Algorithms*, pages 15–28, Italy, Sept. 1990. Springer-Verlag (LNCS 486). To appear in *Theoretical Comp. Sci.*

[4] Y. Afek, S. Kutten and M. Yung. Local Detection for Global Self Stabilization In *Theoretical Computer Science*, No 186, pp. 199-229. 1997.

[5] B. Awerbuch, B. Patt-Shamir and G. Varghese. Self-stabilization by local checking and correction. In *Proc. 32nd IEEE Symp. on Foundation of Computer Science*, 268–277, 1991.

[6] B. Awerbuch, B. Patt-Shamir, G. Varghese and S. Dolev. Self-stabilization by local checking and global reset. In *Proc. 8th International Workshop on Distributed Algorithms*,

[Awe85] B. Awerbuch. "Complexity of Network synchronization". *Journal of the ACM*,32(4):804-823, octobre 1985.

[BCD95] J. Beauquier, S. Cordier et S. Delaët. "Optimum probabilistic self-stabilization on uniform rings". *Proceedings 2th Workshop on Self-Stabilizing Systems*, Las Vegas, mai 1995.

[BD95] J. Beauquier et O. Debas. "An optimal self-stabilizing algorithm for mutual exclusion on bidirectional non uniform rings". Dans *WSS95*, pp 17.1-17.13, 1995.

[BGK98] J. Beauquier, C. Genolini and Shay Kutten. "k-Stabilization of Reactive Tasks". *POD'C98 Proceedings of the* $17^{th}$ *Annual ACM Symposium on Principles of Distributed Computing*, 1998.

[BGK99] J. Beauquier, C. Genolini and Shay Kutten. "Optimal Reactive k-Stabilization : the case of Mutual Exclusion" *POD'C99 Proceedings of the* $18^{th}$ *Annual ACM Symposium on Principles of Distributed Computing*, pages 209–218, Mai 1999.

[BLB95] F. Butelle, C. Lavault et M. Bui. "A uniform self-stabilizing minimum diameter tree algorithm" Dans *WDAG95*, pp. 257-272, 1995.

[BP89] J.E. Burns et J. Pachl. "Uniform self-stabilizing rings". *Transactions of programming languages and systems*, 11(2):330-344, avril 1989.

[CYH91] Nian-Shing Chen, Hwey-Pyng Yu et Shing-Tsaan Huang "A self-stabilizing algorithm for constructing spanning trees". *Information Processing Letters*, 39:147-151, North-Holland, 1991. pages 209–218, Mai 1999.

[Del95] Sylvie Delaët. "Auto-stabilisation : Modèle et Application à l'Exclusion Mutuelle". Thèse, Université de Paris-Sud, France 1995.

[DGS96] S. Dolev, M.G. Gouda et M. Schneider. "Memory requirements for silent stabilization". dans *PODC96*, pages 27-34, 1996.

[DGT00] A. Datta, M. Gradinariu et S. Tixeuil. "Self-stabilizing Mutual Exclusion Using Unfair Distrituted Scheduler".

[DH95] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. In *Proc. of the Second Workshop on Self-Stabilizing Systems*, pages 3.1–3.15, May 1995.

[Dij65] E. W. Dijkstra. "Solution of a problem in concurrent programming control". *Communication of ACM*,8(9):569, Septembre 1965.

[Dij74] E. W. Dijkstra. "Self-stabilizing systems in spite of distributed control". *Communications of ACM*, 17(11):643-644, novembre 1974.

[DIM93] S. Dolev, A. Israeli et S. Moran. "Self-stabilizing of Dynamic Systems Assuming Only Read/Write Atomicity". *Distributed Communications of ACM ? ou Distributed Computing ?*, Vol.7, PP.3-16, 1993.

[Dol00] S. Dolev. *Self-Stabilization.* Ed. The MIT Press, Cambridge.

[DT00] Bertrand Ducourthial et Sébastien Tixeuil "Self-stabilization with Path Algebra". Dans *Proceedings of the Seventh International Colloquium on Structural Information and Communication Complexity (SIROCCO2000)*, pp. 95-110, Italie, juin 2000.

[FD94] M Flatebo et AK Datta. "Two-state self-stabilizing algorithms for token rings". Dans *IEEETSE*, vol. 20, pp. 500-504, 1994.

[FDG94] M Flatebo, AK Datta et S Ghosh. "Self-stabilization in distributed systems" Dans *Readings in Distributed Computing Systems*, IEEE Computer Society Press, pp 100-114, 1994.

[FDS94] M Flatebo and AK Datta and AA Schoone", "Self-stabilizing multi-token rings" Dans *Distributed Computing*, vol. 8, pp. 133-142, 1994.

[FLBB82] M. J. Fischer, N. A. Lynch, J. E. Burns and A. Borodin. "Ressource allocation with immunity to limited process failure". *20th Annual Symposium on Foundations of Computer Sciences*,page 234-254, San juan, Octobre 1979.

[Gen97] C. Genolini. "Un algortihme réparti de partitionnement des graphes" *RenPar9*, mai 1997 à Lausanne, Suisse.

[Gen00] C. Genolini. "Optimal k-stabilisation : the case of synchronous Mutual Exclusion" Dans *PDCS2000*, Las Vegas, USA, Novembre 2000

[GGHP96] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemamraju. "Fault-containing self-stabilizing algorithms". Dans *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, Philadelphie, Pennsylvanie, USA, Mai 1996.

[GH96] M. Gouda and F. Furman Haddix. "The Stabilizing Token Ring in Three Bits". *Journal of Parallel and Distributed Computing 35*, pages 43–48, 1996.

[GHS83] Robert G. Gallager, P.A. Humblet et P.M. Spira. "A distributed algorithm for minimum-weight spanning trees". *ACM Transactions on Programming Languages and Systems*, 5(1):66-77, Janvier 1983.

[GKP93] Juan A. Garay, Shay Kutten et David Peleg. "A sub-linear time distributed algorithm for minimum-weight spanning trees". *34th Annual Symposium on Foundations of Computer Science*, pages 659-668, Palo Alto (Californie), Novembre 1983.

[GlL77] Gérard Le Lann. "Distributed Systems- towards a formal approach". Ed. Bruce Gilchrist, 1977.

[Her90] T. Herman. "Probabilistic self-stabilization". *Information Processing Letters 35*, pages 63-67, 1990.

[Her91] T. Herman. "Adaptativity through Distributed Convergence". *Thèse*, Août 1991.

[HW95] LC Wuu et ST Huang. "Distributed self-stabilizing systems". Dans *Journal of Information Science and Engineering*, vol. 11, pp 307-326, 1995.

[IJ90] Amos Israel et Marc Jolfon. "Token management schemes and random walks yield self-stabilizing mutual exclusion". *Proceeding 9th Principles of Distributed Computing*, page 119-129, 1990

[IJ93] Amos Israel et Marc Jolfon. "Uniform self-stabilizing ring orientation". *Information and Computing*,104:175-196, 1993.

[JADT99] Colette Johnen, L.O. Alima A.K. Datta et S. Tixeuil. "Self-stabilizing neighborhood synchronizer in tree networks". Dans *ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems*, pp487-494, 1999.

[JdR94] J. de Rumeur (ouvrage collectif). "Communication dans les Réseaux de Procésseurs". Ed. Masson, 1994.

[KP93] S. Katz and K. Perry. Self-stabilizing extensions for message-passing systems. In *Distributed Computing*, Vol. 7, pages 17-26, 1993.

[KP97] Shay Kutten and Boaz Patt-Shamir. "Time-adaptive self-stabilization". *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 149–158, Aug. 1997.

[KP98] Shay Kutten et Boaz Patt-Shamir. "Asynchronous Time-Adaptive Self Stabilization". Dans *Proceedings of ACM PODC* (papier court) 1998.

[KP95] Shay Kutten and D. Peleg. "Fault-local distributed mending". *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, août 1995.

[KY97] H. Kakugawa et M. Yamashita. "Uniform and self-stabilizing token rings allowing unfair daemon". *IEEE Transactions on Parallel and Distributed System"*, (8):154-162, 1997.

[Lam83] L Lamport. "Solved problems, unsolved problems and non-problems in concurrency, invited address" Dans *PODC84*, pp 63-67, 1983.

[Lav95] C. Lavault. *Evaluation des Algorithmes Distribués.* Ed. Hermes, 1995

[Lyn96] N.A. Lynch. *Distributed Algorithms.* Ed. Morgan Kaufmann, 1996.

[Pey76] Peyo. "La Schtroumpfette". Ed. Dupuis.

[Pey78] Peyo. "52 Histoires de Schtroumpfs". Ed. Dupuis.

[Ray85] M. Raynal. *Algorithmes Distribués et Protocoles.* Ed. Eyrolles, 1985.

[Ray86] M. Raynal. *Algorithms for Mutual Exclusion.* Ed. MIT Press, Cambridge, 1986.

[RH90] M. Raynal et J-M Helary. *Synchronization and control of Distributed Systems and Programs.* Ed. John Wiley and Sons, Chichester, Angleterre, 1990.

[Sch93] Marco Schneider. *Self-stabilization.* ACM Computing Surveys, volume 25(1), pages 45-67, mars 1993.

[SG89] J.M. Spinelli et R.G. Gallager. "Event driven topology broadcast without sequence number". *IEEE Transaction on Communication*, 37(5):468-474, Mai 1989.

[Tel94] G. Tel. *Distributed Algorithms.* Ed. Cambridge University Press, 1994.

[Tix00] Sébastion Tixeuil. "Auto-stabilisation Efficace". Thèse, Université de Paris-Sud, France 2000.

[TH94] M.S. Tsai et S.T. Huang. "A self-stabilizing algorithm for the shortest paths problem with a fully distributed demon" Journal of Parallel Processing Letters, volume 4, pages 65-72, 1994.