Marches aléatoires et mot circulant, adaptativité et tolérance aux pannes dans les environnements distribués.

Thibault Bernard

Université de Reims Champagne-Ardenne
Ecole Doctorale Sciences Technologies et Santé

Thèse

présentée par **Thibault BERNARD**

pour l'obtention du grade de

Docteur de l'Université de Reims Champagne-Ardenne

Spécialité : Informatique

Marches aléatoires et mot circulant, adaptativité et tolérance aux pannes dans les environnements distribués.

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

Président	Pr Ivan Lavallée	Professeur à l'Université de Paris 8
Rapporteurs	Pr Pascal Felber	Professeur à l'Université de Neuchâtel (Suisse)
	Pr Vincent Villain	Professeur à l'UPJV
Examinateurs	Pr Olivier Flauzac	Professeur à l'URCA
	Pr Herwig Unger	Professeur à l'Université de Hagen (Allemagne)
Directeur	Pr Alain Bui	Professeur à l'URCA

Centre de Recherche en STIC équipe Systèmes Communicants

"Mais il doit être évident que tout ce que j'ai fait, a été pour moi une manière de vivre, et non un moyen pour survivre."
Walter Bonatti

Table des matières

Table des figures

Liste des Algorithmes

Introduction

Le développement des réseaux informatiques, l'apparition des technologies sans fil, l'hétérogénéité des supports de communication et des périphériques de calcul et le comportement imprévisible des utilisateurs accroissent les occurrences de pannes lors des échanges d'informations au sein de ces systèmes. Un système informatique réparti ne peut plus à l'heure actuelle négliger l'impact de pannes ou d'erreurs. Celles-ci peuvent malheureusement se traduire par des pertes considérables. Il est donc nécessaire de développer des solutions permettant de gérer au mieux les défaillances du système.

L'utilisation de tels systèmes est motivée par, entre autres, le partage d'informations et de ressources et l'augmentation de performances. Cette tendance s'est accrue avec le GRID Computing (Globalisation des Ressources Informatiques et des Données, introduit dans [FK99] avec le projet Globus). Cette nouvelle technologie consiste à gérer un ensemble d'ordinateurs pour combiner, entre autres, leur puissance de calcul et donc résoudre des problèmes de taille considérable. Ces ressources de calcul sont en général hétérogènes et d'un degré de performance fort disparate. Le développement de telles applications doit donc prendre en compte cette hétérogénéité.

Le gain en fiabilité est lui aussi une nécessité dans les réseaux où l'indisponibilité d'un service ou d'un composant peut entraîner de fâcheuses conséquences. Ce problème est d'autant plus récurrent dans les réseaux dynamiques. Les réseaux ad-hoc et les réseaux pair-à-pair entrent dans cette catégorie : l'évolution des composants du système est conditionnée par leur état, leur position et leur évolution géographique. Néanmoins la prédiction de cette évolution dépend de critères extérieurs qui sont difficilement analysables à cause de la diversité de leur origine. Il convient donc de concevoir des protocoles de communication qui soient résistants à ces reconfigurations géographiques. On peut citer comme applications à de tels réseaux, les applications militaires, le déploiement de capteurs de surveillance, les systèmes d'échange de fichiers, ou encore les applications décentralisées de calcul intensif.

La conception de solutions pour de tels réseaux est rendue difficile du fait de reconfigurations imprévisibles. En effet, pour échanger des informations, les composants du système peuvent utiliser deux types de solutions : un envoi de message via un chemin prédéfini (à travers des structures de contrôle), ou bien une inondation du réseau avec ce message. Dans les deux cas, il n'y a aucune garantie du bon acheminement du message sous un schéma de mobilité arbitraire. C'est pourquoi la conception de solutions dans de tels environnements requiert des hypothèses sur la mobilité de chacun des composants.

Nos travaux se positionnent entre l'algorithmique distribuée et les réseaux dynamiques : nous proposons des solutions pour les réseaux dynamiques aux problèmes classiques des systèmes distribués (en particulier les problèmes de structuration et d'allocation de ressources). Nous avons choisi comme modèle de réseau dynamique, un modèle général (graphe dynamique) afin de ne pas restreindre la portée de nos travaux à un type particulier de réseaux dynamiques. Contrairement aux systèmes distribués classiques, l'occurrence de reconfigurations topologiques dans un réseau dynamique ne peut plus être considérée comme *"exceptionnelle"*, néanmoins elles ne sont pas forcément considérées comme une évolution *"naturelle"*.

La gestion de ces reconfigurations est généralement effectuée à l'aide d'algorithmes qui maintiennent une structure de communication virtuelle. Ces solutions s'adaptent rapidement aux reconfigurations topologiques, néanmoins un certain délai est requis pour prendre en charge celles-ci. Ce délai peut aussi s'accroître du fait de défaillances transitoires survenant sur le réseau. A notre avis, la gestion de ces reconfigurations devrait s'effectuer à l'aide d'outils qui gèrent naturellement le dynamisme du réseau pour se focaliser sur la gestion des défaillances transitoires.

Nous proposons donc une solution alternative : l'utilisation des marches aléatoires comme outil pour l'écriture d'algorithmes distribués dans le contexte des réseaux dynamiques. Dans [DS00], les auteurs comparent une marche aléatoire, à la promenade d'un ivrogne sur Madison avenue. Ils proposent d'interpréter le théorème de Pólya[1] à l'aide d'une analogie entre le comportement d'une marche aléatoire et celui d'un réseau électrique formé de résistances unitaires. Une marche aléatoire peut être vue, dans le cas de l'algorithmique distribuée, comme le déplacement aléatoire d'un message[2] dans le réseau.

Les marches aléatoires sont adaptées aux environnements dynamiques. En effet, leur politique de déplacement ne nécessite aucune connaissance globale de la structure du réseau pour diffuser une information à chacun des composants du système. Mais, comme nous l'avons précisé précédemment, aucune stratégie ne peut garantir la visite d'un noeud du réseau par un message : il est toujours possible de trouver un adversaire qui utilise le dynamisme du réseau pour contrer cette stratégie (cf. CMPS+04). Les marches aléatoires n'échappent pas à cette règle. Néanmoins, elles permettent de s'affranchir de la gestion du dynamisme du réseau.

En outre l'utilisation des marches aléatoires nous permet de concevoir des algorithmes totalement décentralisés[3], critère à notre avis indispensable dans un réseau dynamique. Dans le cas contraire, l'éviction d'un site ayant un rôle prépondérant, amènerait de nouvelles difficultés.

[1] *"a random walker on an infinite street network in d-dimensional space is bound to return to the starting point when $d = 2$, but has a positive probability of escaping to infinity without returning to the starting point when $d = 3$."*

[2] communément appelé "jeton".

[3] C'est-à-dire que l'algorithme est le même pour chacun des sites. Certains auteurs parlent aussi d'algorithmes uniformes(cf. [BP89, KY97]), mais cette définition n'est pas reconnue de manière unanime.

Dans un premier temps, nous nous sommes intéressés à l'évaluation de la complexité d'algorithmes fondés sur des marches aléatoires. Cette évaluation s'effectue en calculant les valeurs caractéristiques des marches aléatoires. [Tet91, CRR+97] ont aussi utilisé l'analogie entre marches aléatoires et réseaux électriques pour évaluer les grandeurs caractéristiques d'une marche aléatoire. Nous avons élaboré, à partir de ces travaux, une méthode qui permet un calcul automatique de ces valeurs caractéristiques. Ces travaux font l'objet du chapitre 3.

Le deuxième aspect de nos travaux concerne la conception d'algorithmes distribués pour les réseaux dynamiques. Nous proposons l'utilisation combinée des marches aléatoires et du mot circulant pour concevoir des algorithmes tolérants aux pannes dans les réseaux dynamiques. Un mot circulant (cf. [Lav86]) est un message collectant des informations au travers de ses déplacements dans le réseau. Ces informations collectées nous permettent la gestion et la correction des défaillances transitoires. Nous avons utilisé dans le chapitre 4 le mot circulant pour maintenir une image reconfigurable de la structure du réseau. Cette image peut alors être distribuée à chacun des composants du système ou être utilisée à la demande, par ceux-ci. Notre technique a l'avantage d'être permanente (aucune intervention ou mécanisme extérieur est requis dans l'occurrence d'une reconfiguration du réseau) et peu coûteuse (à chaque instant, il n'y a qu'un seul message en transit).

Dans le chapitre 5, nous avons utilisé cette structure reconfigurable pour écrire un algorithme auto-stabilisant[4] de circulation de jeton. Le problème de circulation de jeton tolérante aux pannes a été étudié sous différents aspects ([Var94, IJ90, BGJDL02, DJPV00]). Nous nous sommes placés dans le cadre d'un algorithme écrit dans le modèle de passage de messages. Nous avons donc résolu le problème de circulation de jeton dans un environnement décentralisé et dans une topologie de réseau arbitraire. La difficulté réside principalement dans la résolution du problème d'interblocage de communication[5] : comment s'assurer alors de l'absence de jeton dans les conditions que nous nous sommes imposés.

Nous avons adapté dans le chapitre 6, cette circulation de jeton aux réseaux ad-hoc. La conception d'algorithmes dans ce type d'environnement nécessite en effet une prise en compte accrue de l'hypothèse de mobilité des composants du système. Plusieurs solutions ([CW05, WWV01]) proposent de construire et de maintenir une structure virtuelle afin que chaque composant effectue ses opérations à l'aide d'une structure de contrôle globale. Notre approche est différente : les marches aléatoires permettant une couverture du graphe en temps fini, nous utilisons cette propriété et le mot circulant pour proposer une solution alternative pour le problème d'allocation de ressources dans les réseaux ad-hoc.

[4]Un système est dit auto-stabilisant ([Dij74]) si, à la suite d'une défaillance transitoire, il est capable de retrouver un comportement satisfaisant les spécifications du problème. L'utilisation de l'auto-stabilisation permet d'éviter les gestions spécifiques des pannes et les "reprises sur erreurs".

[5]Les composants du systèmes attendent des messages, mais aucun message n'est en transit sur le réseau.

Chapitre 1

Systèmes répartis et modèles de calcul

Résumé : *Nous introduisons dans ce chapitre quelques rappels sur les systèmes distribués. Nous présentons leur définition et les modèles de calcul couramment rencontrés dans la littérature. Nous détaillons ensuite un panel non exhaustif des problèmes classiques de l'algorithmique distribuée.*

1.1 Présentation et définition d'un système distribué

Dans [Tel94], un système distribué est défini comme étant *" une collection de ressources autonomes de calcul interconnectées"*. Chacune de ces entités est appelée *site* quel que soit sa nature (ordinateurs, processus, processeurs, ...).

Pour être qualifiées d'*autonomes*, l'auteur précise *"que ces ressources doivent assurer leur propre contrôle."*, ainsi une machine parallèle de type SIMD (Simple Instruction Multiple Data) n'entre pas dans la catégorie des systèmes distribués. La qualification d'*interconnectées* signifie que les ressources peuvent s'échanger des informations (quelque soit le support de communication).

L'algorithmique distribuée a pour but l'implantation répartie d'algorithmes "classiques" augmentés de primitives de communication : émission et réception d'informations. Chaque site va, par le biais de communications, effectuer un calcul local et aboutir à l'élaboration d'un résultat global et ce, souvent de manière concurrente.

L'étude des systèmes distribués consiste donc à étudier d'une part l'architecture de communication entre les différents sites et d'autre part à concevoir des algorithmes distribués et analyser leurs performances.

1.1.1 Modélisation d'un système distribué

Un système distribué (ou réseau) est modélisé par un graphe en général non orienté $G = (V, E)$ où V représente l'ensemble de sites du système distribué et E l'ensemble des liens de communications entre les différents sites. Nous utilisons donc dans la suite de nos travaux la terminologie issue de l'algorithmique distribuée et de la théorie des graphes

indifféremment. Nous rappelons quelques définitions élémentaires et leurs significations en algorithmique distribuée.

Définition 1.1 *Soit i et j, deux sites, i et j sont dits voisins si et seulement si $(i,j) \in E$. Le site i peut donc communiquer directement avec le site j.*

Définition 1.2 *Le voisinage du site i (noté $Vois_i$) est l'ensemble des sites qui sont voisins de i. C'est l'ensemble des sites avec lesquel i peut communiquer directement.*

Définition 1.3 *Un chemin existe entre deux sites i_1 et i_n, s'il existe une séquence $i_2, \ldots, i_{n-1}$ telle que : $(i_j, i_{j+1}) \in E$ avec $1 \leq j < n$. C'est à dire que le site i_1 peut communiquer avec le site i_n via la séquence de sites $i_2, i_3, \ldots, i_{n-2}, i_{n-1}$.*

Définition 1.4 *Un chemin $i_1, i_2, \ldots, i_{n-1}, i_n$ est dit simple si et seulement si $\forall (j,k) \in [1, ..., n]$ avec $j \neq k$, on a $i_j \neq i_k$.*

Définition 1.5 *Un réseau est dit connexe s'il existe un chemin entre tout couple (i,j) de sommets. C'est à dire que le réseau d'interconnexion permet le dialogue entre deux sites arbitrairement choisis qui ne sont pas nécessairement voisins.*

1.1.2 Problématique de l'algorithmique distribuée

L'un des problèmes majeurs en algorithmique distribuée est l'absence de vision globale des éléments composant le système. Chaque site a une vision de ses données locales et éventuellement de son voisinage mais n'a la possibilité de consulter les données ou l'état des autres sites que via le mécanisme de communication. L'utilisation de ce mécanisme de communication entre les deux sites engendre un délai dans le temps de calcul et donc le site i réclamant une information sur le site j, peut avoir à la réception de celle-ci, une vision déjà périmée de l'information contenue sur le site j.

Un autre problème majeur concerne le coût d'une communication. En effet, une solution évidente à un problème tel que l'acheminement d'une information d'un site i vers un site j serait d'envoyer cette information à tous les sites voisins (mécanisme d'inondation). A la première réception de cette information, un site la renvoie à tous ses voisins excepté celui qui lui a transmis. Ainsi de proche en proche, le site j finit par recevoir cette information. Le coût de cette communication est donc $O(m)$ messages (m étant le nombre de liens de communication du réseau). Il est pourtant possible, en sélectionnant le plus court chemin entre i et j d'acheminer cette information avec un coût inférieur ou égal à $n-1$ messages (la longueur du plus grand chemin simple entre i et j, n étant le nombre de sites du réseau). Or dans un système distribué, les sites s'échangent en permanence des informations et l'utilisation d'une stratégie d'acheminement de messages inadaptée surchargera alors considérablement la bande passante du réseau et conduira inévitablement à une baisse importante des performances du système.

L'exécution d'un algorithme distribué est non-déterministe : l'ordre de réception des messages et la disparité des délais de communication sont sources d'aléas. Si deux sites i et j envoient au même moment un message au site k, l'ordre de réception de ces deux

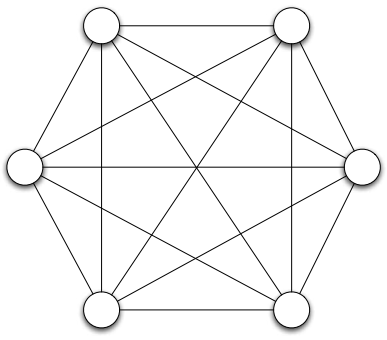
messages peut modifier le comportement du site k et donc l'exécution future de l'algorithme. Même si elles fournissent un résultat satisfaisant les spécifications du problème, deux exécutions consécutives d'un algorithme sur un même réseau peuvent donc fournir des résultats différents.

Enfin, l'ensemble des sites composant le système distribué n'ont pas la même perception du temps : il n'existe pas d'horloge globale dans un système distribué. Chaque site ne peut évaluer l'état d'avancement des autres sites que par le biais des communications qu'il a eu avec ces autres sites, communications qui sont asynchrones. Néanmoins, nombre de solutions considèrent l'existence d'une horloge globale ou au moins de communications synchrones (c'est-à-dire que le délai de communication est borné).

1.2 Organisation des systèmes distribués

Les systèmes distribués peuvent être structurés de manière quelconque (c'est-à-dire que les interconnexions entre les différents sites du système ne respectent aucune règle particulière) ou être organisés en fonction d'un certain nombre de règles. Certains algorithmes ne fonctionnent qu'en exploitant la particularité topologique d'un système distribué. Nous détaillons ici les topologies les plus couramment utilisées pour la conception d'algorithmes.

Clique

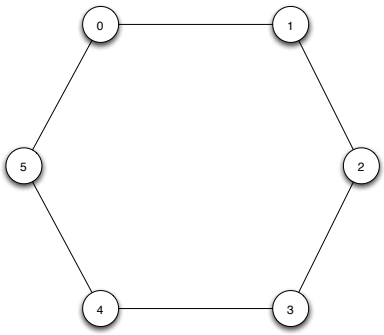


Clique de 6 noeuds

Définition 1.6 *Une clique, ou réseau complet, est un réseau où pour tout couple de sites (i, j), i est voisin de j.*

Dans ce type de topologie, tous les sites peuvent communiquer entre eux de manière directe et non à travers une séquence de sites. La mise en place d'une stratégie permettant la réduction de communications y est donc particulièrement aisée.

Anneau

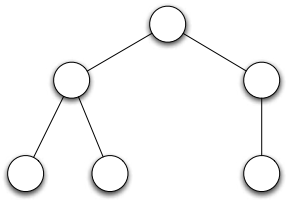


Anneau de 6 noeuds

Définition 1.7 *Un anneau de taille n est un réseau où les sites numérotés de* 0 *à* $n-1$ *sont tels qu'il n'existe un canal de communication qu'entre i et $i+1 \mod n$*

Il existe exactement entre deux sites distincts quelconques deux chemins simples.

Arbre

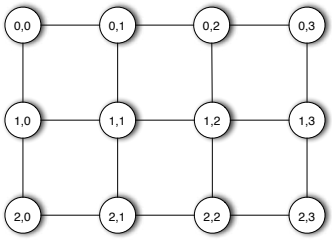


Arbre de 6 noeuds

Définition 1.8 *Un arbre de n noeuds est un réseau connexe contenant exactement $n-1$ liens de communication.*

Un arbre a la particularité de ne pas contenir de cycle. Il existe alors entre deux sites qu'un unique chemin simple. Il est possible de construire un arbre couvrant sur tout réseau arbitraire connexe.

Grille



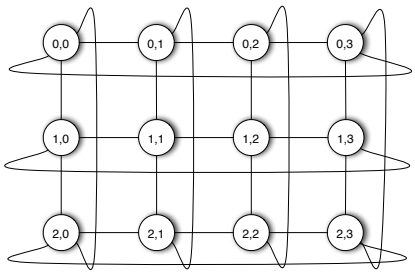
Grille de 12 noeuds

Définition 1.9 *Une grille de $n \times m$ noeuds est un réseau pour lequel chaque noeud est étiqueté (i,j) tel que $i < n, j < m$. Deux noeuds (i,j) et (i',j') sont voisins si :*

$$((i=i')\wedge(j=j'+1))\vee((i=i')\wedge(j=j'-1))$$

$$\vee((i=i'+1)\wedge(j=j'))\vee((i=i'-1)\wedge(j=j'))$$

Grille torique



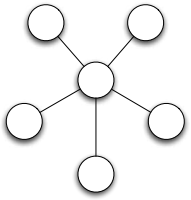
Grille torrique de 12 noeuds

Définition 1.10 *Une grille torique de $n \times m$ noeuds est un réseau pour lequel chaque noeud est étiqueté (i, j) tel que $i < n, j < m$. Deux noeuds (i, j) et (i', j') sont voisins si :*

$$((i = i') \wedge (j = j' + 1 \mod n))$$
$$\vee((i = i') \wedge (j = j' - 1 \mod n))$$
$$\vee((i = i' + 1 \mod n) \wedge (j = j'))$$
$$\vee((i = i' - 1 \mod n) \wedge (j = j'))$$

Tous les sommets d'une grille torique ont 4 voisins.

Etoile

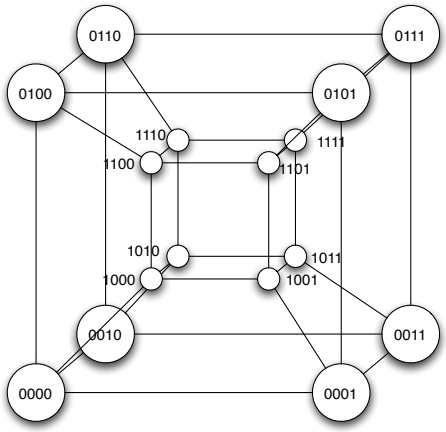


Etoile de 6 noeuds

Définition 1.11 *Une étoile est un arbre où un noeud (appelé le centre) est voisin de tous les autres sommets.*

La distance maximale entre deux sites dans une étoile est de 2.

Hypercube



Hypercube de 16 noeuds

Définition 1.12 *Un hypercube est un graphe $HC_N = (V, E)$ avec $N = 2^n$ sommets (n est appelé la dimension de l'hypercube). Ici V est un ensemble de chaîne de bits de longueur n :*

$$V = \{b_0, \ldots, b_n\}, b_i \in \{0, 1\},$$

et a et b sont voisins si et seulement si les chaînes a et b diffèrent exactement d'un bit.

La distance entre deux sites i et j est exactement le nombre de bits différents entre les chaînes i et j. Elle est donc d'au plus n. Cette topologie est beaucoup utilisée dans les machines parallèles.

1.3 Les différents modèles de calcul

Nous présentons dans cette section, les différents modèles de calcul qui représentent le fonctionnement d'un système distribué. Nous commençons par détailler les différents modèles de communication entre les sites interconnectés du réseau. Puis nous introduisons les différents modèles d'exécution qui sont considérés lorsqu'un algorithme s'exécute sur un réseau de sites.

1.3.1 Modèles de communication

Un système distribué est constitué d'un ensemble de sites qui par le biais d'algorithmes, établissent de manière concurrente, un résultat global. L'exécution de ces algorithmes est conditionnée par la manière dont les différents sites communiquent. Nous détaillons ici trois modèles de communication :

Le modèle à états

Ce modèle a été introduit par [Dij74] pour le problème de circulation de jeton autostabilisante dans un réseau structuré en anneau. Dans ce modèle, chacun des sites composant le système adopte un *état* qui représente l'état des différentes données du site. Les communications ne se font pas via des échanges de messages, mais chaque site peut directement consulter l'état des sites voisins.

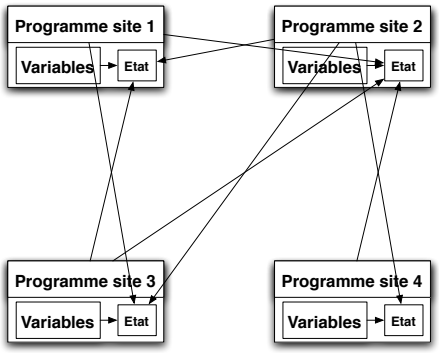


FIG. 1.1 – Modèle à états

L'écriture d'algorithmes dans le modèle à états se présente sous la forme de règles gardées :

$$< Garde > \longrightarrow < Actions >$$

Les gardes s'écrivent sous la forme d'une expression booléenne et dépendent de l'état du site qui déclenchera la règle ainsi que de celui de ses voisins. Les actions modifient l'état du site sur lequel elles sont exécutées.

Le modèle à registres

Dans ce modèle, les communications s'effectuent par le biais de registres qui sont associés à chaque canal de communication. Chaque canal possède deux registres qui sont accessibles uniquement aux sites adjacents à ce canal. Par exemple, le canal (i, j) possède 2 registres : $R_{i,j}$ et $R_{j,i}$. Le site i peut alors accéder en lecture/écriture au registre $R_{i,j}$ et en lecture seulement au registre $R_{j,i}$. L'écriture d'algorithmes se fait de la même manière que pour le modèle à états mais en ne considérant que les registres concernés.

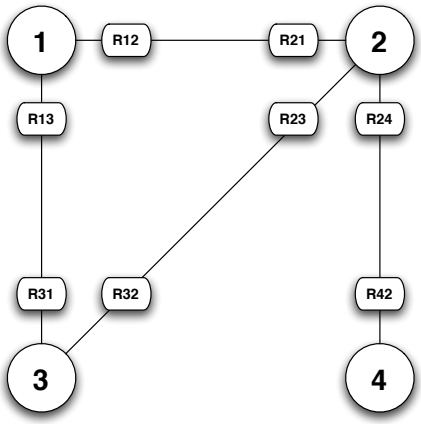


FIG. 1.2 – Modèle à registres

Le modèle à passage de messages

Dans ce modèle, les communications s'effectuent par le biais de messages échangés entre deux sites. Les hypothèses sur les canaux de communication peuvent être diverses, les canaux peuvent être de taille bornée[1] ou non, FIFO[2] ou non. L'écriture d'algorithmes est cette fois ci basée sur la réception de messages : celle-ci peut déclencher un certain nombre d'événements (mise-à-jour de variables locales et/ou envoi de nouveaux messages).

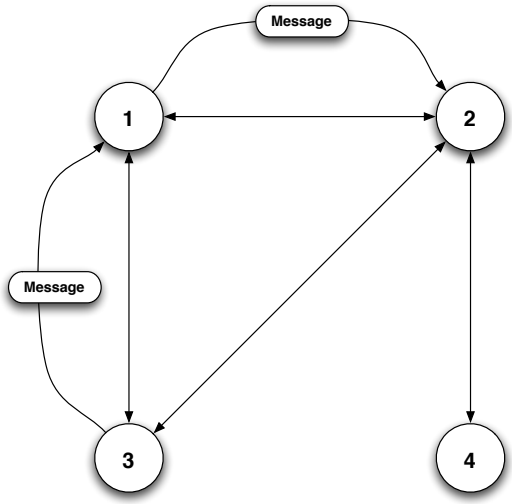


FIG. 1.3 – Modèle à messages

1.3.2 Modèles d'exécutions

L'exécution d'un algorithme distribué peut se modéliser par un système de transition qui représente les évolutions du système.

Définition 1.13 *Un système de transition est un triplet* $S = \{C, \longrightarrow, I\}$ *où :*
- C *est un ensemble de configurations*
- $\longrightarrow$ *est une relation de transition binaire sur* C
- I *est un sous-ensemble de* C *représentant les configurations initiales.*

Définition 1.14 *Une exécution est une séquence maximale* $E = (\gamma_1, \gamma_2, \ldots)$ *où* $\gamma_0 \in I$ *et* $\forall i, \gamma_i \longrightarrow \gamma_{i+1}$

Définition 1.15 *Une configuration* $\gamma \in C$ *est dite terminale s'il n'existe pas de configuration* α *telle que* $\gamma \longrightarrow \alpha$.

Définition 1.16 *Un algorithme est dit* permanent *s'il n'existe pas de configuration terminale.*

[1] L'écriture devient alors bloquante si le canal est déjà rempli. La reception du message libère alors de la place sur le canal.

[2] Premier message envoyé, premier message reçu.

Un algorithme peut alors se définir via ses configurations et ses relations de transition.

Nous allons maintenant regarder les solutions qu'offre la littérature sur les problèmes classiques de l'algorithmique distribuée.

1.4 Algorithmes classiques

La conception d'applications réparties dans un système distribué nécessite l'utilisation d'algorithmes de contrôle distribué. Ces algorithmes forment des primitives de base pour l'écriture d'une application répartie. Nous pouvons citer entre autres :

1.4.1 Election

Le problème de l'élection a pour la première fois été posé et résolu en 1977 par LeLann [LeL77]. Ce problème consiste, à partir d'une configuration où tous les sites sont *candidats*, à atteindre une configuration où un site est déclaré *leader* et tous les autres sont déclarés *battus*.

Le problème de l'élection peut être ramené à un problème de consensus où l'ensemble des sites doivent s'accorder sur une valeur : l'identifiant du site à élire. Dans [FLP85], il est montré que le consensus distribué n'a pas de solution dans un système asynchrone si au moins un site peut devenir défaillant.

La solution de LeLann s'utilise dans un environnement non fautif structuré en anneau. La complexité en message de cet algorithme est en $O(n^2)$. Chang et Roberts ont amélioré cet algorithme [CR79] en bloquant les messages contenant des identités déjà battues : la complexité moyenne en message est en $O(n \log n)$. Ces algorithmes ont encore été améliorés par [Pet82, DKR82], en maximisant la complexité en message par $O(n \log n)$. D'autres solutions ont été proposées dans d'autres types de réseaux. Dans [KKM90], il est montré qu'il existe une relation entre l'élection et le parcours d'un réseau (construction d'un arbre par exemple).

1.4.2 Diffusion d'informations

La diffusion d'informations dans un réseau consiste à envoyer une information d'un site à tous les autres. Ce procédé peut être employé pour nombre de tâches : envoi d'un ordre, synchronisation, calculs locaux, ... Après la diffusion du message, chaque site peut éventuellement prendre localement une *décision*.

La plupart des algorithmes de propagation d'informations se basent sur le principe de parcours intégral du graphe d'interconnexion. Par exemple, l'algorithme d'inondation procède au parcours intégral des arêtes du graphe d'interconnexion. Tous ces algorithmes sont *des algorithmes à vagues*.

Définition 1.17 *Un algorithme à vagues est un algorithme qui satisfait les critères suivants :*

Terminaison *Toute exécution est finie.*

Décision *Toute exécution contient au moins un événement de Décision.*

Dépendance *Dans toute exécution, chaque événement de Décision est causalement précédé par un autre événement dans chacun des sites.*

De par la définition, la complexité en message d'un algorithme à vagues dans un réseau arbitraire, est d'au moins $|E|$ messages.

Les algorithmes PIF (Propagation d'Informations avec Retour, cf. [Seg83]) sont des algorithmes à vagues. L'objectif de ces algorithmes est de propager une information et d'en faire le retour jusqu'au site initiateur.

Les algorithmes *de parcours* (cf. [Tel94]) sont des algorithmes à vagues pour lesquels tous les événements sont *totalement ordonnés* (c'est-à-dire qu'il n'y a qu'un seul message en transit à un instant donné). Dans [Tar95], l'auteur propose un algorithme de parcours qui construit un arbre sur le réseau.

1.4.3 Structuration

Dans un réseau arbitraire, un site ne peut envoyer directement des messages qu'aux sites qui lui sont adjacents (son voisinage). Pour réussir l'acheminement d'un message entre deux sites non voisins, ce message devra traverser un chemin rejoignant les deux sites. L'objectif de la structuration est de fournir à chaque site un moyen d'acheminer les messages reçus vers chacune de leurs destinations finales. Une solution classique est le calcul de tables de routage. Ces tables locales aux sites indiquent un chemin possible pour atteindre chacune des destinations finales. Le routage est donc composé d'une part, du calcul des tables de routage, et d'autre part de l'acheminement des messages. Les critères pour un *bon* routage sont [Tel94] :

Correction Les procédures mises en oeuvre doivent assurer l'acheminement des messages vers leurs destinations finales.

Efficacité Les messages doivent être acheminés le plus rapidement possible et à travers le chemin le plus court possible.

Complexité Le calcul des tables de routage doit s'effectuer avec une complexité en message et en temps la plus basse possible.

Robustesse Dans le cas où une reconfiguration topologique se produit, l'algorithme doit recalculer les tables de routage afin de conserver le critère de *correction*.

Adaptativité L'algorithme doit répartir au mieux la charge des messages afin de prévenir la surcharge de certains canaux de communication.

Equité Le service doit être disponible de manière uniforme pour chacun des sites.

Tous les critères ne peuvent être satisfaits. Par exemple, un algorithme de routage qui s'adapte très rapidement à la charge du réseau, aura une complexité plus importante qu'un algorithme qui n'adapte pas ses tables de routage. Il faut alors trouver un compromis en fonction des tâches à réaliser sur le système.

L'optimalité d'un routage dépend de ce qu'est un bon routage. Tel cite plusieurs notions d'un *bon* routage :

Minimum de sauts Le nombre d'arêtes que devra traverser un message émis par i à destination de j doit être minimum.

Plus courts chemins Un coût (positif ou nul) est assigné pour la traversée de chacune des arêtes du graphe. Il s'agit ici de minimiser le coût des communications entre les sites, donc de trouver le chemin entre le site i et j dont le coût est le moindre. Le calcul en *Minimum de sauts* est une version particulière du *plus courts chemins* : le coût pour traverser une arête est unitaire.

Délai minimum Sur chaque canal de communication, un poids est ajusté dynamiquement en fonction du trafic sur ce canal. Un algorithme de routage basé sur cette notion, cherchera à ajuster les tables de routage de manière à minimiser les délais de transmission.

L'algorithme de [Tou80] permet le calcul des tables de routage en plus courts chemins, mais ne tolère pas les reconfigurations topologiques du réseau d'interconnexion. L'algorithme de [MS79] calcule pour chaque destination, des arbres en largeur d'abord et autorise les changements topologiques. Enfin, l'algorithme de [Taj77] calcule des tables de routage en plus courts chemins qui seront remises à jour après un changement topologique.

1.4.4 Allocation de ressources

L'allocation de ressources ou le problème d'exclusion mutuelle se pose quand un ensemble de sites ont besoin de partager l'accès à une ressource. Chaque site peut avoir besoin d'exécuter un morceau de code en section critique, c'est-à-dire, qu'un seul site peut accéder à la ressource partagée à un instant donné. Chaque site comporte alors trois états :

- Etat demande de section critique
- Etat en section critique
- Sortie

La solution au problème d'allocation de ressources est de concevoir un algorithme capable de faire passer un site de l'état demande à l'état section critique en vérifiant les propriétés de sûreté et de vivacité.

Sûreté A tout instant il y a au plus un site dans l'état section critique

Vivacité Tout site qui se trouve dans l'état demande finit par passer dans l'état section critique. Il n'y aura donc pas de problème d'interblocage ou de famine grâce à la propriété de vivacité.

Dans [Ray91b], les algorithmes d'exclusion mutuelle ont été classés en deux catégories, d'une part les algorithmes basés sur les permissions, où chaque site, pour accéder à la section critique, doit en obtenir la permission auprès des autres sites, et d'autre part les algorithmes basés sur l'utilisation d'un jeton circulant. La possession de ce jeton symbolise le privilège d'accéder à la ressource critique.

Dans [RA81], les auteurs présentent une solution basée sur les permissions. Chaque site demande à tous les autres s'il est autorisé à entrer en section critique. Si tous les sites répondent favorablement, le site peut alors entrer en section critique. Une fois le code en section critique exécuté, le site rend les permissions à chacun des sites. [CR83] proposent une amélioration : à la fin d'une section critique, un site ne rend pas les permissions, il les garde jusqu'à une nouvelle demande de section critique.

L'autre approche, à base de jeton, a une complexité fixe à chaque instant, au plus un message est en transit : le jeton. L'algorithme de Lelann [LeL77], précurseur de cette approche, propose un jeton circulant perpétuellement sur un anneau de processeurs Cette approche a été reprise dans [Mar85]. L'unicité du jeton assure la propriété de sûreté de l'algorithme. La circulation du jeton à travers tous les sites composant le système, garantit que la propriété de vivacité sera respectée. Dans [IJ90] les auteurs proposent une solution fonctionnant sur tous types de topologies : un jeton se déplaçant comme une marche aléatoire (cf. Section 2.5). Les propriétés de couverture de la marche aléatoire assurent le respect de la propriété de vivacité.

1.5 Conclusion

Nous avons défini de manière générale un système distribué, les différents composants qui le constituent, la structuration de ces composants et nous avons présenté les différents modèles de calcul rencontrés dans la littérature. Ensuite, nous avons brièvement exposé une partie des principaux problèmes classiques de l'algorithmique distribuée. Il existe évidemment bien d'autres types de problèmes dans les environnements distribués, nous pouvons, entre autres, citer *la détection de terminaison* (La configuration globale est terminale, mais l'état de chacun des sites ne l'est pas forcément, cf. Section 2.4 pour une définition du problème et un exemple de solution), *la synchronisation* (obtenir une horloge globale sous certaines conditions), *le problème des réseaux anonymes* (les différents sites du système ne sont pas distingables), ou encore *la tolérance aux pannes*, sur laquelle nous reviendrons (Section 2.3).

Chapitre 2

Problématique et outils

Résumé : *Nous détaillons dans ce chapitre le contexte spécifique de nos travaux : la conception de solutions tolérantes aux pannes dans les réseaux dynamiques. Nous introduisons la notion de* mot circulant*, outil de collecte et de dissémination d'informations sur un réseau et nous présentons le concept de marche aléatoire que nous utilisons comme modèle de circulation pour le mot circulant.*

2.1 Introduction

Nous nous sommes intéressés dans nos travaux à l'élaboration d'algorithmes de contrôle tolérants aux pannes dans les réseaux dynamiques. Contrairement aux systèmes distribués "classiques", un réseau dynamique est en général modélisé par un graphe qui évolue au fur et à mesure du temps. Cette évolution peut parfois être prévisible dans certains types de réseaux, mais en général aucune hypothèse ne peut être faite sur les reconfigurations topologiques du graphe de communication.

L'élaboration de solutions à des problèmes généraux, dans les réseaux dynamiques, doit, à notre avis, nécessairement se faire à travers l'écriture d'**algorithmes décentralisés**[1], c'est-à-dire que le code de l'algorithme doit être le même sur chacun des différents sites du système. En effet, le dynamisme du réseau peut mener à l'éviction d'un site qui possède un rôle prépondérant dans le fonctionnement de l'algorithme. Pour la même raison, les algorithmes de contrôle distribué déployés sur des réseaux dynamiques ne doivent pas tenir compte de la particularité de la topologie.

Néanmoins, la conception d'algorithmes de contrôle distribué tolérants aux pannes dans les réseaux dynamiques n'admet pas, sous un schéma de mobilité arbitraire, de solution qui soit absolument fiable. En effet, il est toujours possible de trouver un adversaire (cf. [PSM+04]) qui utilise le dynamisme du réseau pour "contrer" le bon déroulement de ces algorithmes. La plupart de ces algorithmes sont donc écrits en faisant des hypothèses sur la mobilité de chacun des composants du système.

Nous avons, dans nos travaux, utilisé les marches aléatoires comme schéma de parcours pour dériver des algorithmes de contrôle distribué dans les réseaux dynamiques. En effet, le parcours d'une marche aléatoire sur un réseau n'utilise qu'une "connaissance locale" du

[1] Certains auteurs parlent aussi d'algorithmes uniformes (cf. [BP89, KY97])

réseau d'interconnexion, ce qui en fait un outil particulièrement bien adapté aux réseaux dynamiques dès lors que ce réseau[2] reste connexe. Il reste néanmoins vrai qu'un adversaire omniscient peut, à l'aide du dynamisme, contrecarrer le parcours de la marche aléatoire.

Nous avons aussi utilisé un mot circulant (*i.e.* un message qui transite sur le réseau afin de collecter de l'information) pour obtenir une image partielle de la topologie du réseau d'interconnexion. Cette image est utilisée non pas pour influencer le parcours de la marche aléatoire, mais pour permettre éventuellement dans le cas de corruption de données d'assurer le caractère tolérant aux pannes de nos algorithmes.

Nous structurons donc ce chapitre en détaillant notre contexte de travail, c'est à dire les réseaux dynamiques puis la tolérance aux pannes. Puis nous présentons les outils que nous utilisons : le mot circulant et les marches aléatoires.

2.2 Réseaux dynamiques

Nous comprenons dans réseaux dynamiques les réseaux qui sont fréquemment sujet à une reconfiguration topologique du graphe de communication, c'est à dire que les ensembles V et E de G ne sont plus statiques, de nouveaux noeuds peuvent apparaître, d'autres disparaître, l'ensemble des arrêtes peut lui aussi évoluer dans le temps. Cela comporte aussi bien les réseaux filaires dont les sites sont sujets à des pannes franches qui peuvent dans ce cas être assimilées à des déconnections de sites, que les réseaux sans fil, où l'ensemble des composants sont mobiles, et donc les canaux de communication entre deux composants évoluent en fonction de la distance séparant ces composants. Si la conception d'algorithmes prenant en charge de tels changements topologiques est devenue dorénavant un enjeu majeur en algorithmique distribuée, c'est aussi devenu un challenge intellectuellement stimulant.

Dans [BKP03], les auteurs classifient les réseaux sans fil en deux catégories :

Les réseaux cellulaires sont caractérisés par l'existence d'une infrastructure fixe où les communications s'effectuent via un réseau filaire jusqu'à un périphérique qui fait office d'interface entre la partie filaire du réseau et la partie comprenant les périphériques sans fil. C'est par exemple le cas des téléphones portables et des ordinateurs disposant d'une carte sans fil. Pour ces réseaux, il est donc suffisant de proposer un protocole de communication sans fil entre deux composants, la gestion de l'infrastructure pouvant s'effectuer à l'aide des algorithmes distribués classiques.

Les réseaux ad-hoc sont des réseaux complètement décentralisés et hautement dynamiques. Il n'y a donc plus d'infrastructure fixe, chaque noeud peut se déplacer de la zone de transmission d'un noeud à une autre, modifiant ainsi le graphe de communication.

Ce qui séparent clairement les réseaux ad-hoc d'autres types de réseaux dynamiques c'est d'une part la fréquence des changements topologiques qui affectent le réseau et d'autre part, la variabilité des délais de communication.

[2] comprendre ici le graphe sous-jacent

Modèles de mobilité

Nous présentons différents modèles de mobilité. Nous avons utilisé ces modèles pour simuler nos algorithmes (cf. Section 6.4). Dans [HGPC99], les auteurs proposent pour effectuer des simulations une modélisation de la mobilité d'un réseau ad-hoc (appelée Reference Point Group Mobility[3]) : chaque groupe de périphériques a un centre logique et le mouvement de celui-ci va définir le mouvement de tout le groupe. Ce modèle permet de générer les positions physiques des noeuds du réseau, mais ne permet pas de distinguer les noeuds uniquement par leur positions physiques : si plusieurs groupes se superposent sur une même région, il n'y a aucun moyen de les distinguer. Dans [WL02], les auteurs étendent le modèle précédant et proposent un autre modèle (Reference Velocity Group Mobility) où chaque périphérique est représenté par un vecteur de vélocité. Un état de l'art sur les modèles de mobilité est proposé dans [CBD02]. Les auteurs présentent quelques modèles de mobilité :

Modèle de mobilité basé sur une marche aléatoire Chaque périphérique se déplace comme une marche aléatoire en prédéfinissant les intervalles de distance [Dav00].

Modèle de mobilité basé sur des points aléatoires Proche du précédant, il prévoit une pause entre chaque changement de direction et de vitesse d'un périphérique [JM96a].

Modèle de mobilité basé sur une direction aléatoire Dans ce modèle, chaque périphérique va se déplacer dans une direction choisie au hasard, jusqu'à atteindre les bornes définies par la simulation. Une fois ces bornes atteintes, le périphérique attend un certain temps avant de choisir une nouvelle direction et de se déplacer jusqu'à atteindre une nouvelle borne [RMSL01] .

Modèle de mobilité Gauss-Markov Ce modèle est une version aléatoire du précédent, le déplacement dépend des déplacements précédents mais aussi d'une valeur aléatoire : la valeur de la vitesse et la direction à l'instant n sont calculées à partir des valeurs à l'instant $n-1$ et d'une valeur α :

$$s_n = \alpha s_{n-1} + (1-\alpha)\bar{s} + \sqrt{(1-\alpha^2)s_{n_{x-1}}}$$

$$d_n = \alpha d_{n-1} + (1-\alpha)\bar{d} + \sqrt{(1-\alpha^2)d_{n_{x-1}}}$$

où, s_n et d_n sont la vitesse et la direction à l'instant n, α est le paramêtre aléatoire compris entre 0 et 1, $\bar{s}, \bar{d}$ sont les valeurs moyennes de s et d quand $n \to \infty$ et $s_{x_{n-1}}, d_{x_{n-1}}$ sont les variables aléatoires d'une distribution de Gauss [LH03, Tol99].

Modèle de mobilité basé sur la topologie d'une ville Dans le cas de ce modèle, la région simulée est une ville et les noeuds sont inclus dans des véhicules. Les règles qui vont régir le réseau d'interconnexion entre les périphériques, correspondent donc aux comportements des véhicules et aux infrastructures de la ville (autoroute, limitation de vitesse, feux,...)[Dav00]

Dans [CBD02], d'autres modèles de mobilité sont aussi présententés, des modèles où le choix des déplacements dépend aussi des autres périphériques :

[3]Mobilité d'un groupe à partir d'un point de référence

Modèle de mobilité basé sur une communauté nomade Dans ce modèle, les périphériques vont se déplacer comme des enfants dans un musée. Ils suivent les déplacements d'un point de référence (le guide) en s'organisant de manière aléatoire autour de celui-ci. La différence entre ce modèle et le précédant réside dans le temps écoulé avant le rassemblement des périphériques autour du point de référence [SM01].

Modèle de mobilité basé sur une poursuite Ce modèle peut s'utiliser lorsque les périphériques traquent une cible particulière (des policiers cherchant un criminel dans une course poursuite) [SM01].

Modèle de mobilité à partir d'un point de référence est le modèle que nous avons décrit au début de la section [HGPC99].

Le modèle de mobilité permet à partir des rayons de communications de chacun des périphériques de calculer leurs voisinages et donc de reconstituer le graphe de communication du réseau.

Algorithmes distribués sur un réseau ad-hoc

Pour la conception d'algorithmes distribués dans les réseaux ad-hoc, il existe deux visions la première qui consiste à ne faire que très peu d'hypothèses sur la mobilité des nœuds et donc à s'intéresser principalement aux problèmes de communication et la seconde qui consiste à résoudre les problèmes classiques de l'algorithmique distribuée mais nécessitant des hypothèses plus fortes sur la mobilité des noeuds.

Nous allons dans le cadre de nos travaux nous intéresser à la deuxième vision, celle qui consiste moyennant quelques hypothèses à concevoir des algorithmes résolvant des problèmes classiques d'algorithmique distribuée sur des réseaux ad-hoc.

Routage en réseau ad-hoc Beaucoup d'algorithmes écrits pour les réseaux ad-hoc utilisent un procédé d'inondation. Ce procédé consiste à la première réception du message à émettre celui-ci à travers tous ses canaux. Comme le précisent les auteurs de [CDCD05], ce procédé est extrêmement coûteux et peut parfois être inadapté, puisqu'il surcharge considérablement la bande passante et donc augmente considérablement le coût en énergie. Ils proposent en solution alternative pour le problème du routage, l'utilisation d'un *paquet nomade de contrôle* qui se déplacera en suivant la trajectoire d'une marche aléatoire. Ce paquet transfert les informations de routage (les arrêtes les moins surchargées, les plus courts chemins) au cours de sa traversée du réseau et met à jour les informations contenues localement dans chacun des périphériques. La problématique de conception d'algorithmes pour les réseaux ad-hoc est bien là : chercher l'optimalité n'a aucun sens puisque le réseau est dynamique et donc la moindre reconfiguration topologique entraîne inévitablement un nouveau calcul des routes et donc affecte les performances du système. Il faut donc concevoir des algorithmes extrêmement légers, qui tolèrent provisoirement un état obsolète, mais qui garantissent un retour à un état cohérent rapidement. Les auteurs classifient les algorithmes de routage pour réseaux ad-hoc en deux catégories :

Le routage proactif construit et maintient des routes à l'aide de mises à jour périodiques des tables de routage en chacun des sites : routage par des agents mobiles

[BDDN01], Routage adaptatif par construction d'arbre enraciné sur la source [Per00] ou encore sa version probabiliste [BDF01].

Le routage réactif consiste à construire les routes à la demande, c'est à dire qu'un périphérique avant de communiquer avec un autre, devra au préalable calculer un chemin jusqu'à sa cible : Routage Préemptif [GAGPK01], Routage avec équilibrage de charge [HZ01] ou encore Routage par sources dynamiques [JM96b].

Les auteurs de [CDCD05] stipulent que le routage dans les réseaux ad-hoc devrait être un routage réactif : selon eux, le coût d'un routage proactif est prohibitif. Ceci, à notre avis, est à nuancer. Suivant la fréquence d'envoi de messages entre périphériques, un routage réactif peut s'avérer plus coûteux qu'un routage proactif *"léger"*.

Allocation de ressources en réseau ad-hoc Les auteurs de [CW05] proposent une solution auto-stabilisante au problème d'allocation de ressources dans un réseau ad-hoc. Leur solution est basée sur un jeton circulant. La circulation de celui-ci est basée sur l'algorithme LRV[4]. Le schéma de circulation converge vers un anneau virtuel, ce qui permet, à l'aide d'un site distingué, de savoir si un jeton a été perdu. Lors d'un changement de topologie, l'anneau virtuel va être reconstruit à l'aide de l'algorithme LRV. La propriété de sûreté est garantie quelque soit le critère de mobilité appliqué. La propriété de vivacité est garantie sous certaines hypothèses (un périphérique hors de portée de tous les autres périphériques ne pourra pas émettre le jeton à l'un de ses voisins).

Cet article met en évidence un critère fondamental pour la conception d'algorithmes auto-stabilisants : l'occurence d'une reconfiguration topologique dans un réseau ad-hoc ne doit pas être assimilée à une panne mais doit au contraire être considérée comme un comportement normal du système. La difficulté de concevoir des algorithmes auto-stabilisants dans de tels environnements vient du fait qu'un changement topologique ne doit pas *corrompre* le système et l'amener dans un état illégal. Cela montre pourquoi, il n'est pas possible d'utiliser les algorithmes distribués classiques dans un réseau ad-hoc et pourquoi la plupart des algorithmes conçus pour les réseaux ad-hoc requierent de respecter quelques contraintes de mobilité.

2.3 Tolérance aux pannes dans les systèmes répartis

L'occurrence d'une faute dans un système peut provoquer l'effondrement de celui-ci. Au vue de l'accroissement du nombre de composants dans un système, la tolérance aux pannes est devenue un enjeu fondamental dans la conception des systèmes répartis.

La tolérance aux pannes doit garantir le bon fonctionnement du système malgré les dysfonctionnements des différents composants du système. Nous présentons les deux approches de la tolérance aux pannes dans un environnement distribué, puis nous détaillons plus particulièrement l'auto-stabilisation.

[4]Least recently visited : le moins récemment visité en français

2.3.1 Généralités

Une faute désigne une défaillance qu'elle soit temporaire ou définitive, d'un ou plusieurs composants du système.

Dans [Ray91a, Tel94, AH93], les fautes ont par exemple été classées en fonction de plusieurs critères :

Cause : Les causes des différentes fautes sont classées en deux catégories : fautes d'omission et fautes byzantines (ou fautes bénignes et fautes malignes). Les fautes bénignes sont simplement des défaillances de composants. Par contre lorsqu'un composant subit une faute byzantine, son comportement ne répond plus à la spécification qui le définit.

Origine : Quelle est la nature du composant qui subit un dysfonctionnement ?

Détectabilité : La faute peut-elle être détectée localement (c'est-à-dire sans l'émission et la réception de messages) ?

Durée : La faute est-elle temporaire ou définitive ?

[Tel94] classifie les fautes pouvant affecter les sites de la manière suivante :

Site mort-né Un site est mort-né, s'il n'exécute aucune action de son algorithme local.

Site en panne Un site est dit en panne, s'il s'est exécuté correctement jusqu'au moment ou il stoppe son exécution.

Site byzantin Un site est dit byzantin, s'il effectue des actions qui ne concordent pas avec son algorithme local. Il peut en particulier envoyer des messages au contenu arbitraire.

Cette classification ne prend pas en compte les fautes qui affectent d'autres composants comme les canaux de communication ou les messages en transit sur le réseau.

Dans [MW87], il est montré que dans un environnement tolérant aux pannes, la plupart des problèmes décrits Section 1.4, se réduisent au problème du consensus (tous les participants doivent s'accorder sur une valeur). Dans [FLP85], Fischer, Lynch et Patterson montrent que le problème du consensus n'admet pas de solution déterministe dans un système asynchrone, même dans l'hypothèse où les pannes sont franches (c'est-à-dire sans comportement byzantin).

Pour concevoir des algorithmes tolérants aux pannes, deux approches ont été proposées : la robustesse qui garantit le respect des spécifications du problème malgré l'occurrence de fautes et l'auto-stabilisation qui garantit qu'un système, après l'occurrence d'une faute, retrouvera son comportement normal en un temps fini.

La conception d'algorithmes robustes consiste principalement à résoudre le problème du consensus avec des hypothèses plus restrictives que le cas du réseau asynchrone [DDS87]. Dans [CT91] les auteurs proposent l'utilisation de détecteurs de fautes plus ou moins fiables. [Tel94] précise que l'utilisation de systèmes synchrones avec des détecteurs de pannes peut être envisagée lorsque l'on dispose d'un environnement de programmation distribuée. Il demeure que dans les environnements dynamiques la fiabilité des détecteurs de fautes peut être mise en doute : ceux-ci peuvent parfaitement se déconnecter. De plus nous nous plaçons dans le cas d'un réseau de périphériques hétérogènes qui peuvent ne pas

inclure une couche logicielle garantissant la détection de pannes. C'est pourquoi dans le cadre des réseaux dynamiques, nous nous sommes naturellement intéressés à la conception d'algorithmes auto-stabilisants.

2.3.2 Auto-stabilisation

Ce principe a été introduit par Dijkstra en 1974 dans [Dij74] : l'auteur présente un algorithme auto-stabilisant de circulation de jeton dans un modèle de communication à états. Mais l'incidence de ces travaux n'a été soulignée qu'en 1984 par Lamport dans [Lam84].

L'auto-stabilisation a été développée pour permettre aux systèmes répartis de résister aux fautes transitoires : telle la corruption des données locales et des messages en transit sur le réseau. Le principe de l'auto-stabilisation est qu'un système, après avoir subit une défaillance transitoire, finira par adopter un comportement qui respecte les spécifications du problème.

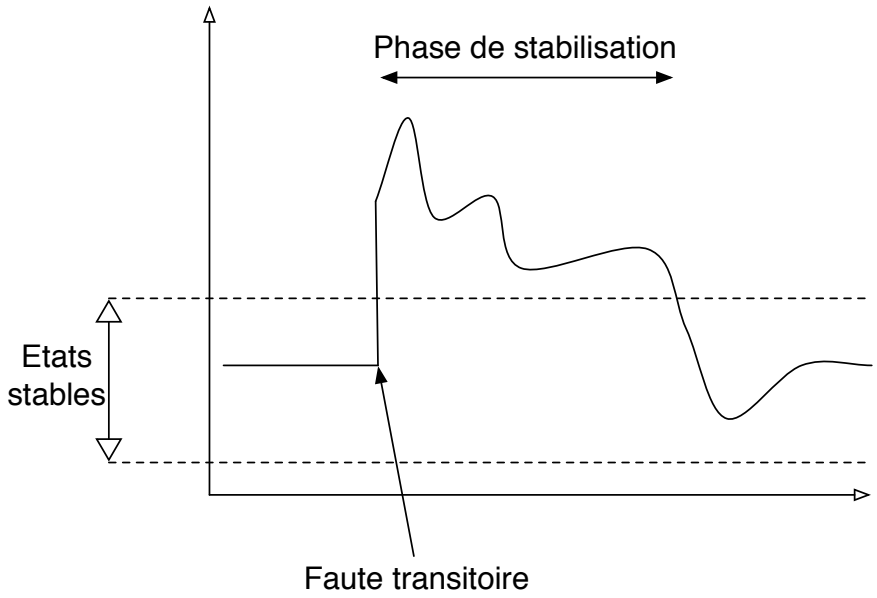


FIG. 2.1 – Principe de l'auto-stabilisation

Afin de prouver qu'un algorithme est auto-stabilisant, il est nécessaire de définir clairement ce qu'est une configuration légale et de prouver deux propriétés [Gou95, AG94] :

Propriété de convergence : sans occurrence de fautes, la configuration de l'algorithme converge vers une configuration légale.

Propriété de clôture : si la configuration du système est une configuration légale et sans l'occurrence de fautes, le système restera à jamais dans une configuration légale.

Pour prouver ces deux propriétés, il est nécessaire de s'appuyer sur la définition de l'état légal.

Remarque 2.1 *Les algorithmes auto-stabilisants corrigent les fautes transitoires, ils n'ont donc pas besoin d'être initialisés.*

Nous présentons maintenant l'algorithme de Dijkstra [Dij74]. Il s'agit d'un algorithme permanent de circulation d'un jeton. Il est écrit dans le modèle à états sous forme de règles gardées (cf. Section 1.3).

Algorithme 1 Algorithme de Dijkstra (74) écrit sous forme de règles gardées

Variables
$\forall i, 1 \leq i \leq n, M_i$ de type booléen
Règles
$\forall i, 2 \leq i \leq n : M_i \neq M_{i-1} \rightarrow M_i \leftarrow M_{i-1}$
$i = 1 : M_1 = M_n \rightarrow M_1 \leftarrow (M_1 + 1) mod K (K > n)$

Nous déroulons ici l'algorithme sur trois sites avec une initialisation arbitraire afin de montrer comment le comportement général de l'algorithme permet de revenir dans un état *légal*.

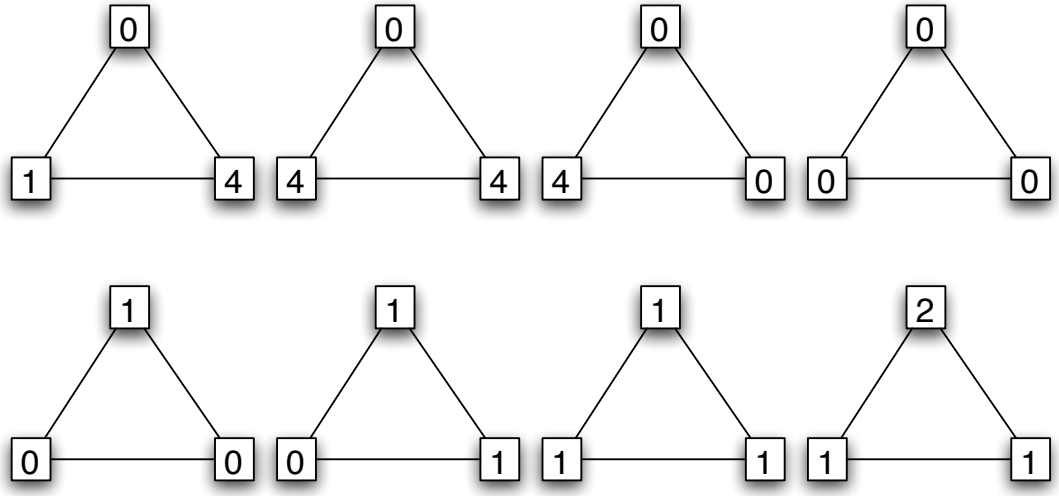


FIG. 2.2 – Exécution de l'Algorithme 1

L'auteur définit l'ensemble des configurations légales $\mathcal{L}$ de la manière suivante : les configurations composant $\mathcal{L}$ sont des configurations où seul un site est déclenchable.

L'idée générale de la preuve est que si $K > n$, à partir d'un certain laps de temps (au plus $\frac{n \times (n-1)}{2}$ étapes), il n'y aura plus qu'un seul site qui sera déclenchable : le site 0. Donc un seul jeton circule sur l'anneau.

Dans [KP93], les auteurs étendent l'auto-stabilisation au modèle à passage de messages asynchrone dans des graphes arbitraires en effectuant un snapshot[5] . Celui-ci leur permet de vérifier que la configuration actuelle du système est légale. Si cette configuration

[5]prise d'instantané en français (*i.e.* récuperation de l'état de chacun des composants du système).

n'est pas légale, les auteurs proposent de réinitialiser chacun des composants du système. Cette méthode permet une auto-stabilisation automatique des différents algorithmes fonctionnant sur des systèmes à passage de messages. Cette méthode a été améliorée dans [FV97] en réduisant considérablement la complexité du snapshot et du retour d'informations en effectuant ces opérations sur l'arbre auto-stabilisant construit dans [PV97]. Dans [APSVD94], les auteurs proposent une alternative au snaphot global : ils effectuent des tests locaux avant éventuellement de réinitialiser le système. Néanmoins, ces algorithmes sont assez coûteux puisqu'il est nécessaire de recourir à un snapshot.

D'autres approches ont été développées pour la conception d'algorithmes auto-stabilisants dans le modèle à passage de messages : concevoir des protocoles de communication auto-stabilisants. [GM91] prouvent que les protocoles de communication auto-stabilisants avec des canaux non bornés ont un nombre d'états infini et doivent utiliser des méchanismes de *timeout*[6]. Dans [AB93], les auteurs proposent un protocole auto-stabilisant de communication point-à-point (Protocole de bit alterné). Celui-ci se base sur l'utilisation d'estampilles et sur la génération de nombres aléatoires et se stabilise dans le cas où les canaux sont FIFO[7] et de taille finie mais non bornée. Dans [HNM99], les auteurs présentent les différentes hypothèses sur les systèmes :

1. Les canaux de communication sont bornés ou non. Gouda et Multari [GM91] ont prouvé qu'un protocole auto-stabilisant utilisant des canaux non bornés a nécessairement un ensemble de configurations légales infini.
2. Les canaux de communication sont fiables ou non. Si les canaux sont considérés comme fiable, la perte de message est considérée comme une faute transitoire. Si le système est un système à canaux bornés, quand un site envoie un message à travers un canal déjà saturé, soit le message est perdu, soit le site est bloqué.
3. L'algorithme utilise un méchanisme de timeout ou non. Dans un système où les sites n'effectuent des actions qu'à la réception d'un message, il est nécessaire de garantir que même s'il n'y a pas de messages en transit dans le réseau, le système ne sera pas en état d'interblocage (*communication deadlock* en anglais), et pourra continuer sa tâche normalement.

Les auteurs de [HNM99] clament que pour rendre le protocole de bit alterné décrit dans [AB93] utilisable en pratique (dans ce cas ici : utiliser une séquence de nombre aléatoire périodique), il faut nécessairement que les canaux de communication soient bornés.

La conception de protocoles auto-stabilisants de communication dans des modèles relativement simplistes n'est pas aisée.

L'auto-stabilisation a aussi été dérivée en d'autres notions :

La stabilisation instantanée a été introduite dans [BDPV99]. Un protocole instantanément stabilisant est un protocole qui se stabilise en 0 étape, c'est à dire qu'il respecte toujours sa configuration légale. Dans cet article, les auteurs proposent un algorithme instantanément stabilisant de propagation d'informations avec retour (PIR) sur un arbre. Leur algorithme est écrit dans le modèle à états et ne tient donc pas compte des problèmes de communications qui ont été soulevés précédemment.

[6] délai de garde en français
[7] Premier Entré Premier Sorti en français

Leur algorithme permet entre autre d'effectuer une réinitialisation globale des composants du système. Ces travaux ont été prolongés dans [BCV04, CDV05, BDV05].

La k-stabilisation introduite dans [BGK98], consiste à considérer qu'il n'y aura pas plus de k sites fautifs. Les auteurs développent deux algorithmes k-stabilisant d'exclusion mutuelle qui se stabilisent respectivement en $O(k^2)$ et en $O(k)$ étapes. Néanmoins, dans un réseau à grande échelle où interagissent des facteurs d'ordre différents, il est particulièrement difficile de borner le nombre de sites fautifs.

Le lecteur pourra se référer à [Sch93, Dol00] pour un état de l'art des différents algorithmes auto-stabilisants.

Dans le cas des réseaux sujets à de fréquentes reconfigurations topologiques, l'auto-stabilisation offre la possibilité de gérer ces reconfigurations comme des fautes transitoires. Ces fautes sont corrigées après un certain temps et le système fonctionne alors correctement sur la nouvelle topologie du réseau d'interconnexion.

2.4 Mot circulant

Le mot circulant a été introduit par Lavallée dans [Lav86]. L'auteur présente cet outil dans le cadre d'une solution à la détection de terminaison. Le problème de la terminaison est un problème majeur de l'algorithmique distribuée. Il consiste en deux tâches :

- Détecter que chacun des processus est à l'arrêt.
- Vérifier que le travail accompli soit conforme au calcul voulu.

En effet, le fait de vérifier que chacun des processus est inactif, est insuffisant pour déterminer si l'algorithme s'est terminé correctement : des processus peuvent être en attente de messages qui leur permettraient de poursuivre et d'achever leur calcul. Ces processus sont alors bloqués et l'algorithme global n'a alors pas fini son calcul. La plupart des solutions consiste donc à visiter chacun des processus et vérifier qu'aucun d'entre eux est en attente de message (c'est-à-dire qu'ils sont "bien" finis). Bien des algorithmes de contrôle distribué utilisent des structures particulières pour mener à bien leurs tâches. La visite des différents processus est souvent effectuée en construisant une structure telle que l'arbre ou l'anneau et en déterminant un type de parcours sur cette structure : vague (cf. [DS80]) ou jeton circulant[8] (cf. [DFG83]).

Après avoir rappelé les conditions nécessaires à la terminaison distribuée dans un réseau de processus arborescent, Lavallée expose comment écrire un algorithme de terminaison dans un graphe orienté sans circuit possédant une source. L'idée générale est qu'il est possible d'établir un pré-ordre sur les différents sommets et donc d'effectuer une diffusion permettant alors une fin "normale" pour chaque processus. Il détaille alors le cas du graphe possédant une source mais aussi des circuits : (en considérant le circuit $X_1, \dots X_q$) *"Tant que X_1 n'est pas fermé (*i.e. en cours de terminaison*), il est susceptible d'envoyer des messages vers X_2 donc par voie de conséquence X_q peut recevoir des messages qui sont la conséquence de ceux envoyés par X_1. X_q peut alors envoyer à X_1 des messages qui sont eux-même la conséquence des précédents. Il y a donc là un risque de bouclage."* L'auteur propose alors un algorithme qui calcule les circuits élémentaires contenus dans

[8] message relayé de processus en processus

le réseau.

L'algorithme proposé par Lavallée introduit et utilise *un mot circulant*. C'est un message qui est répercuté de processus en processus (exactement comme un jeton) et qui collecte au travers de ses déplacements, des informations. Dans [Lav86], celui-ci collecte l'identité des processus et est émis par la source à tous ses descendants : *"Lorsqu'un processus reçoit un mot circulant, celui-ci lui est transmis par l'un de ses prédécesseurs et il contient, outre l'identificateur dudit prédécesseur, les identificateurs de tous les antécédents par lesquels il a transité ; le processus ne retransmettra, le cas échéant, le mot circulant qu'après y avoir adjoint son propre identificateur, en queue, de façon à ce que le mot soit une suite ordonnée d'identificateurs, la dernière lettre du mot identifiant le dernier processus ayant réémis le mot circulant."*

Définition 2.1 *Un mot circulant est un message collectant des informations à travers ses déplacements dans le réseau.*

Le récepteur du mot circulant sera alors informé du chemin parcouru par celui-ci et pourra déterminer s'il fait partie d'un circuit :

Théorème 2.1 *Si un processus y reçoit pour mot circulant AyB (A et B étant des sous-mots), alors :*

1. *y fait partie d'un circuit.*
2. *ce circuit est composé des processus dont les identificateurs figurent dans le sous-mot B*

Lorqu'un processus détecte qu'il fait partie d'un circuit, il ré-expédie le mot circulant à son descendant (défini dans le mot circulant) de manière à avertir tous les processus du circuit. Un processus passe dans l'état *averti* dès lors que tous ses prédécesseurs sont soit *avertis* soit *complets*. Un processus passe dans l'état *complet* lorsque ses prédécesseurs ne lui envoient plus de messages et que celui-ci a achevé son traitement local. Donc si un processus a tous ses prédécesseurs à l'état *complet*, il passera, après avoir effectué ses actions locales, à l'état complet. En collectant de l'information topologique, le mot circulant est non seulement utile pour la détection de cycles mais aussi pour avertir chacun des processus, de la terminaison de ses prédécesseurs.

Nous utiliserons dans nos travaux le mot circulant comme un "centralisateur" d'informations. Il fournit à tout instant, au site qui le détient, une image partielle de la topologie du réseau. Nous présenterons d'autres travaux relatifs au mot circulant dans le chapitre 4.

2.5 Marches aléatoires

2.5.1 Introduction à l'algorithmique probabiliste

L'utilisation de l'algorithmique probabiliste s'impose lorsque les algorithmes déterministes ne fournissent pas de solutions ou ne sont pas satisfaisants soit en termes de complexité soit en termes de qualité de solutions. Les algorithmes probabilistes se classent en deux catégories :

Les algorithmes de type Las Vegas fournissent un résultat exact, mais leur complexité en temps, n'est pas bornée.

Exemple 2.1 (Election) *Soit le problème de l'élection d'un leader parmi n sites candidats. L'algorithme d'élection consiste à éliminer par tours successifs les sites candidats votants. Au début, les n candidats sont votants, et à la fin, le seul candidat qui reste votant est le leader. A chaque tour, chaque candidat votant tire un nombre au hasard compris entre 1 et n, l'algorithme compte alors le nombre t de* 1 *qui ont été tirés. Si* $t = 0$*, un nouveau tour est effectué. Si* $t > 1$*, seuls les sites candidats qui ont tiré un* 1 *restent candidats pour le tour suivant. Si* $t = 1$*, l'algorithme termine, le leader étant celui qui a tiré* 1*. Cet algorithme peut ne jamais finir si deux sites candidats tirent toujours* 1*.*

Les algorithmes de type Monté Carlo terminent forcément mais ne fournissent en général qu'un résultat approché [And98, MR95].

Exemple 2.2 (Calcul de π) *Un calcul approché de* π *est possible en tirant au hasard des points sur un carré (de longueur* $2r$*).*

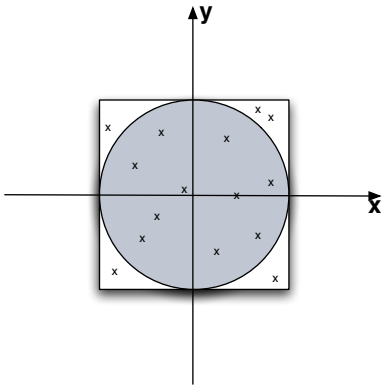


FIG. 2.3 – Calcul probabiliste de π

Une approximation de π *est possible en comparant la surface théorique du disque*(πr^2) *et le ratio du nombre de points inclus dans le disque (vérifiable à partir de l'équation du disque centré sur l'origine* $x^2 + y^2 < r^2$*) et du nombre total des points tirés au hasard.*

En algorithmique distribuée, certains problèmes n'admettent pas de solutions déterministes. Par exemple sur des réseaux anonymes, l'élection d'un leader est impossible : il n'existe aucun moyen de distinguer deux sites de même degré. Il devient alors nécessaire de recourir à des méthodes probabilistes afin de casser la symétrie entre les sites et de pouvoir élire un leader. [IR81] présente une adaptation de l'algorithme de [CR79] aux réseaux anonymes de taille connue : chaque site tire une identité au hasard dans $1, \cdots, n$ avec n le nombre de sites du réseau. Si le site reçoit sa propre identité, cela ne signifie pas qu'il est élu : un autre site à pu choisir lui aussi cette identité. L'idée introduite dans [IR81] est de compter le nombre de sauts du message. Si celui-ci a effectué n sauts c'est

que le message à fait le tour de l'anneau et le site est alors élu. Si le nombre de sauts n'est pas n (*i.e.* au moins deux sites ont tiré la même plus petite identité), chaque site encore candidat procède à un nouveau tour de l'algorithme.

Les marches aléatoires ont été introduites pour les parcours de graphes par [AKL+79, MR95]. Elles ont été dérivées pour écrire des algorithmes distribués : [BIZ89] pour la construction d'un arbre couvrant, [IJ90] pour l'exclusion mutuelle, [VT95] pour l'élection, ... Les algorithmes que nous avons conçus sont des algorithmes de Las Vegas. Leurs temps d'exécution ne sont pas bornés, néanmoins les temps moyens d'exécution sont polynomiaux.

Si l'écriture d'algorithmes distribués fondés sur des marches aléatoires est relativement simple, l'évaluation de la complexité de tels algorithmes l'est moins. A l'heure à laquelle nous avons commencé nos travaux, il n'existait que des bornes sur les grandeurs caractéristiques des marches aléatoires. Ces bornes sont calculables grâce à la théorie de chaînes de Markov.

2.5.2 Définition d'une marche aléatoire

Nous considérons une marche aléatoire au sens de [AKL+79, MR95], c'est-à-dire le déplacement d'un jeton sur un graphe : arrivé en un sommet du graphe, ce jeton choisit un des voisins de manière aléatoire (en général avec une probabilité $\frac{1}{\deg(i)}$ où $\deg(i)$ est le degré du sommet i) et se déplace vers ce sommet.

Cette étape s'appelle un pas de la marche aléatoire. Une marche aléatoire est composée d'une succession de pas.

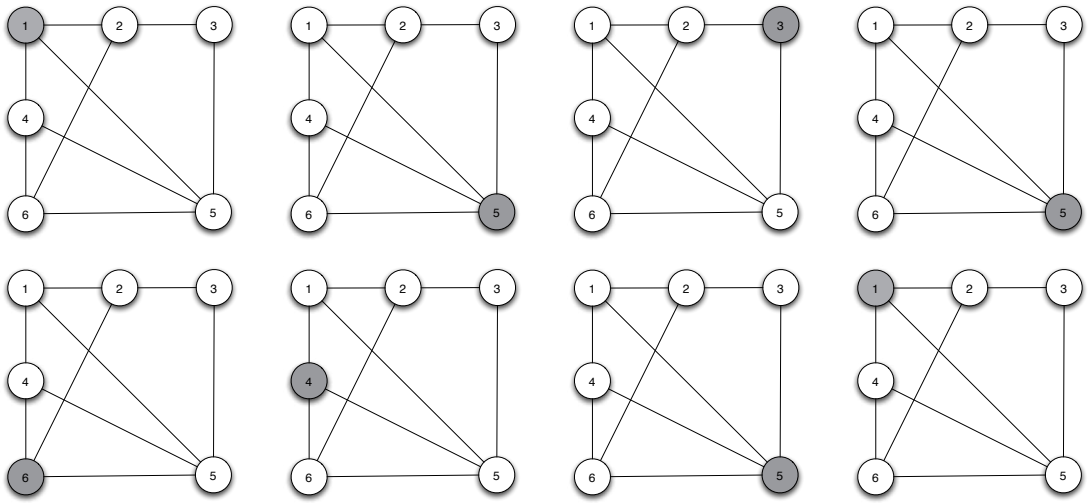


FIG. 2.4 – Quelques pas d'une marche aléatoire sur un graphe

Une marche aléatoire est donc une chaîne de Markov homogène et irréductible sur un graphe non orienté. C'est donc un processus sans mémoire évoluant en temps discret et dans un espace d'états discret. Son état futur ne dépend que de son état présent et non pas des états passés qui l'y ont amenés (cf. [NG98] pour plus de détails).

Une marche aléatoire sur $G = (V, E)$ un graphe non orienté, de n noeuds et m arêtes s'écrit par la donnée de P_{ij} (probabilité que le jeton passe du sommet i au sommet j).

Nous considérons dans la suite de nos travaux, le cas le plus courant : $\forall j \in Vois_i, P_{ij} = 1/\deg(i)$ où $\deg(i)$ est le nombre de voisins du sommet i (degré de i) dans le graphe G. Néanmoins, les résultats généraux restent vérifiés sur un graphe valué. Dans le cadre de notre problématique nous considérerons toujours que $P_{ij} = 1/\deg(i)$ pour j voisin de i. Dans le cas où les arrêtes sont valuées, le lecteur peut se référer à [Soh05].

Il est alors possible d'écrire la matrice des probabilités des transitions P :

$$P = (P(i,j)_{i,j \in V^2})$$

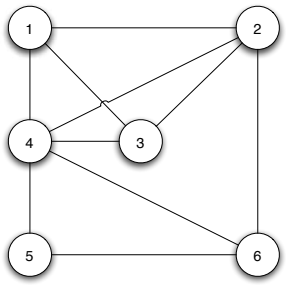
où :

$$P(i,j) = \begin{cases} \frac{1}{\deg(i)}, & \text{si } (i,j) \in E \\ 0, & \text{sinon} \end{cases}$$

La transition de la marche aléatoire se déduit par l'équation suivante :

$$\Pi_{t+1} = \Pi_t \times P$$

Exemple 2.3 *Soit le graphe G suivant aura pour matrice de transition P*



$$P = \begin{vmatrix} 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 & 0 \\ \frac{1}{4} & 0 & \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} & 0 & 0 \\ \frac{1}{5} & \frac{1}{5} & \frac{1}{5} & 0 & \frac{1}{5} & \frac{1}{5} \\ 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} & 0 \end{vmatrix}$$

Le régime permanent Intuitivement le régime permanent n'est atteint qu'après un certain nombre de pas : lorsque la distribution des probabilités de la marche n'évolue plus. Ce régime permanent n'est atteint que sur les graphes non bipartis.

Définition 2.2 *Un graphe est dit biparti, s'il existe deux sous ensembles de sommets tels que dans chacun des sous ensembles, aucun sommet ne soit relié à un sommet de ce même sous ensemble.*

Définition 2.3 *Le régime permanent est atteint lorsque :*

$$\Pi = \Pi \times P$$

Π *s'appelle alors la distribution stationnaire.*

La vitesse avec laquelle la marche aléatoire converge vers la distribution s'appelle *le taux de convergence* (μ). Il se calcule de la manière suivante :

$$\mu = \lim_{t \to \infty} \sup \max_{i,j} \left| p_{ij}^{(t) - \frac{d_j}{2m}} \right|^{1/t}$$

Propriétés des marches aléatoires

Percussion Une marche aléatoire sur un graphe fini, visite infiniment souvent un site arbitrairement choisi.

Couverture Une marche aléatoire sur un graphe fini, visite tous les sommets du graphe en un temps fini mais non borné ([KR93]).

Rencontre Deux marches aléatoires circulant sur un graphe fini, finiront par se rencontrer sur un même sommet (cette propriété est aussi appelée propriété de collision[IJ90]).

Il est évident que dans un système informatique, la seule garantie de ces propriétés n'est pas suffisante. Il est nécessaire de fournir une mesure moyenne en temps partant d'une configuration quelconque du système pour atteindre une configuration où ces propriétés sont respectées.

2.5.3 Grandeurs caractéristiques

Nous parlons ici des grandeurs caractéristiques qui permettent l'évaluation de la complexité en messages d'algorithmes distribués fondés sur des marches aléatoires. Par exemple : en combien de temps, en moyenne, une diffusion est-elle réalisée à l'aide d'une marche aléatoire ? La plupart des bornes des complexités moyennes des algorithmes basés sur des marches aléatoires se calcule à partir des temps de percussion ou des temps de couverture.

Temps de visite ou de percussion

Définition 2.4 *Le temps de visite (noté $h(i,j)$) est le nombre moyen de pas qu'il faut à une marche aléatoire pour, partant d'un site i, atteindre pour la première fois un site j.*

Proposition 2.1 *La valeur du temps de retour ($h(i,i)$) en régime permanent est (le régime permanent est atteint une fois la distribution stationnaire atteinte) :*

$$\frac{1}{\Pi(i)} = h_G(i,i) = \frac{2m}{\deg(i)}$$

où m est le nombre d'arêtes du graphe G et $\deg(i)$ est le degré du sommet i dans le graphe G.

Dans [KS76], les auteurs calculent la matrice des temps de percussion lorsque le régime permanent est atteint :

Propriété 2.1 *Soit $P = P_{ij}$ la matrice des probabilités de transitions associée au graphe G. Alors la matrice des temps de percussion est donnée par la formule suivante :*

$$M = (I - Z + EZ_{dg})D$$

où

- *I est la matrice identité*

- $Z = [I - (P - A)]^{-1}$ *où* $A = a_{i,j} = \lim_{n \longrightarrow \infty} P^n$ *qui est une matrice aux lignes identiques (distribution stationnaire)*
- E *est une matrice dont tous les termes sont égaux à* 1
- Z_{dg} *est la matrice égale à* Z *sur la diagonale est à* 0 *partout ailleurs.*
- D *est la matrice diagonale (*$\frac{1}{a_i}$ *sur la diagonale,* 0 *ailleurs).*

Théorème 2.2 *Pour trois sommets* i, j *et* k*,*

$$h(i,j) + h(j,k) + h(k,i) = h(i,k) + h(k,j) + h(j,i)$$

Temps de commutation

Définition 2.5 *Le temps de commutation (noté* $C(i,j)$*) est le nombre moyen de pas nécessaires à une marche aléatoire partant du site* i *pour atteindre le sommet* j *et revenir au sommet* i*. La valeur de ce temps est :* $C(i,j) = h(i,j) + h(j,i)$

Temps de couverture

Définition 2.6 *Le temps de couverture partant de* i *(noté* $C(i)$ *) est le nombre moyen de pas nécessaires à une marche aléatoire pour visiter tous les sommets du graphe en commençant par le site* i*. Dans le cas général, (*i.e. *partant d'un nœud arbitraire) on le note* C *et* $C = \max_{i \in V} C(i)$*.*

Théorème 2.3 *Bornes de Matthews (cf. [JJLV00])*
Le temps de couverture d'un graphe à n *noeuds est*

- *au plus* $\sum_{i=1}^{n} \frac{1}{n} \times \max_{i,j \in E}(h(i,j))$
- *au moins* $\sum_{i=1}^{n} \frac{1}{n} \times \min_{i,j \in E}(h(i,j))$

Temps de couverture cyclique Le temps de couverture cyclique a été introduit dans [CFS96]. Il se calcule en fonction des temps de percussion et fournit donc une valeur exacte. De plus cette valeur est relativement proche du temps de couverture, puisque la marche aléatoire a nécessairement parcouru tous les sommets si elle a visité tous les sommets dans un ordre donné. Néanmoins, dans [BS05], les auteurs proposent le calcul exact du temps de couverture sur un graphe donné.

Définition 2.7 *Le temps de couverture cyclique est défini comme étant le temps moyen pour visiter tous les sommets dans le meilleur arrangement possible :*

$$CCT = \min \left\{ \sum_{i \in V} h(\sigma(i), \sigma(i+1)) \,/ \sigma \in \mathcal{S}_n \right\}$$

où $\mathcal{S}_n$ *désigne l'ensemble des permutations des identités des sommets du graphe.*

2.5.4 Marches aléatoires multiples

Nous commencons par introduire de manière intuitive les multimarches aléatoires, puis fournissons les différents résultats qui leurs sont associés.

Introduction aux marches aléatoires multiples

Une multimarche aléatoire est composée de plusieurs marches aléatoires circulant sur un même graphe. Les différents jetons sont relayés de sommets en sommets et peuvent éventuellement se rencontrer sur un même sommet. En algorithmique distribuée, ces multi-marches ont été utilisées pour concevoir des algorithmes. Par exemple dans [IJ90], les auteurs les utilisent afin de concevoir un algorithme auto-stabilisant d'exclusion mutuelle. Les rencontres entre les marches aléatoires offrent différentes options intéressantes (échanges ou agrégations d'informations). La principale mesure à calculer est le temps de rencontre entre marches aléatoires, afin de fournir une complexité des algorithmes distribués que nous concevons dans la suite de ce document.

Grandeur caractéristique

Définition 2.8 *Le temps de rencontre entre marches aléatoires est le nombre moyen de pas qu'il faut à plusieurs marches aléatoires $(x_1, \cdots, x_i)$ avant de se rencontrer. On le note $M_G(x_1, \cdots, x_i)$.*

La rapidité avec laquelle ces marches aléatoires vont se rencontrer, dépend de la manière d'ordonnancer leurs déplacements.

Notion de démon

La manière d'ordonner le déplacement des différentes marches aléatoires est dépendante de l'utilisation qui sera faite de la rencontre des marches aléatoires multiples (accélérer la couverture, accélérer l'agrégation d'informations, ...) et bien évidemment du contexte. D'après [TW93], il existe trois manières de gérer les déplacements des marches aléatoires :

L'ordonnancement aléatoire : à chaque pas, un choix est effectué de manière aléatoire. Ce choix détermine laquelle des différentes marches aléatoires va se déplacer. Il n'y a donc ici absolument aucun contrôle sur l'évolution de la couverture du graphe.

Le déplacement simultané : dans ce cas, à chaque pas de la multimarche aléatoire, toutes les marches aléatoires se déplacent (il s'agit en fait du cas synchrone). Encore une fois, il est impossible d'avoir un contrôle sur la rapidité de rencontre entre les différentes marches aléatoires.

Le déplacement décidé par un démon : celui-ci décide à chaque pas, laquelle des marches aléatoires se déplace. Le démon peut mettre en place une stratégie pour permettre une rencontre rapide ou non. Cette stratégie dépend de plusieurs critères (la topologie du réseau par exemple), une stratégie de couverture rapide ne sera pas la même sur des topologies différentes. Le démon de type *"Ange"* choisira les marches dans l'intention de minimiser le temps de rencontre.

Le cas le plus général et le plus facile à mettre en oeuvre dans le cadre de l'algorithmique distribuée est le premier : le déplacement aléatoire. Cet ordonnancement ne fait absolument aucune hypothèse sur le modèle du système. Il est donc possible d'utiliser un système de type asynchrone (le modèle le plus général). En contrepartie, il n'y a aucune méthode permettant d'accélérer (ou retarder) la rencontre entre plusieurs marches aléatoires. Cela nécessite donc d'étudier attentivement les résultats des temps de rencontre entre plusieurs marches afin d'être en mesure de valider ou non une solution basée sur des multimarches.

Résultats sur les temps de rencontre

Dans [IJ90], les auteurs fournissent une borne supérieure au temps de rencontre entre deux marches aléatoires :

Théorème 2.4

$$M_G(u,v) \leq O\left((\Delta-1)^{D-1}\right)$$

où Δ est le degré maximum, et D le diamètre.

Cette borne est exponentielle et n'est pas satisfiable. Heureusement, il ne s'agit pas d'une borne fine et dans [TW91], les auteurs montrent que le temps de rencontre est borné par une fonction polynomiale :

Théorème 2.5 *Le temps de rencontre entre deux marches aléatoires sur le graphe G est borné par :*

$$M_G(u,v) \leq \frac{8}{27}n^3$$

Cette borne a été affinée successivement. Dans [CTW93], la borne sur le temps de rencontre entre deux marches aléatoires est réduite à :

Théorème 2.6

$$M_G(u,v) \leq \frac{4}{27}n^3$$

Dans [TW93], les auteurs donnent des encadrements des temps de rencontre en fonction des ordonnancements effectués :

Théorème 2.7 *Le temps de rencontre entre deux marches aléatoires est borné par :*

Pour deux marches ordonnées de manière aléatoire :

$$\frac{1}{27}n^3 \leq M_G(u,v) \leq \frac{4}{27}n^3$$

Pour deux marches se déplaçant simultanément :

$$\frac{2}{27}n^3 \leq M_G(u,v) \leq \frac{16}{27}n^3$$

Pour deux marches ordonnées à l'aide d'un démon :

$$\frac{4}{27}n^3 \leq M_G(u,v) \leq \frac{4}{27}n^3$$

Pour deux marches ordonnées à l'aide d'un démon de type *Ange* :

$$\frac{1}{27}n^3 \leq M_G(u,v) \leq \frac{4}{27}n^3$$

Dans [BHWG99], une écriture des marches aléatoires est proposée : celles-ci peuvent s'écrire comme un vecteur de la même dimension que le nombre de marches aléatoires sur le graphe et chaque composante contient la position de chacune des marches aléatoires. Ce vecteur s'appelle le vecteur configuration.

Lemme 2.1 *Une configuration* $< y_1, \cdots, y_k >$ *est une configuration suivante de* $< x_1, \cdots, x_k >$ *si* $y_j \in \{V(x_j) \cup x_j\}$.

Calcul en général Dans [BHWG99] le résultat suivant est montré :

Théorème 2.8 *Pour un graphe G non orienté et connexe, pour chaque configuration* $< x_1, \cdots, x_k > \in V^k$ *et pour* $2 \leq l \leq k$ *on a :*

$$M_G(x_1, \cdots, x_k) \leq \frac{1}{l}\sum_{j=1}^{k} M_G(x_j, x_{j+1}, \cdots, x_{j+l-1})$$

Cas particulier de l'anneau Dans [BHWG99], les auteurs développent les résultats sur l'anneau :

Théorème 2.9 *Sur un anneau R, pour des positions cycliquement ordonnées* $< x_1, \cdots, x_k >$ *avec une distance entre les jetons successifs de* $d_1, d_2, \cdots d_k$ *:*

$$M_R(x_1, \cdots, x_n) = \sum_{i<j} (d_i \times d_j)$$

2.5.5 Marches aléatoires et systèmes distribuées

Les marches aléatoires avant d'être utilisées en algorithmique distribuée ont d'abord été utilisées sur des graphes. Dans [AKL+79, KR93], les auteurs prouvent les propriétés de couverture et de percussion d'une marche aléatoire sur un graphe. Dans [Bro89], l'auteur propose un algorithme permettant la génération d'un arbre couvrant aléatoire sur un graphe arbitraire. La construction de l'arbre couvrant est effectuée en centralisant des informations topologiques sur une particule se déplaçant aléatoirement dans le graphe. L'auteur montre que l'arbre construit est choisi uniformément au hasard parmi l'ensemble des arbres couvrants sur le graphe.

Dans [BIZ89], les auteurs proposent, cette fois dans un cadre distribué, un algorithme de construction d'arbre couvrant aléatoire basé au préalable sur l'élection probabiliste d'un

leader. Les auteurs précisent que le choix de structures aléatoires augmente la tolérance aux pannes de celle-ci. En effet des utilisateurs malicieux ne pourront pas prédire (par la connaissance qu'ils ont des algorithmes déterministes de construction d'arbres couvrants) quels sont les liens de communication appartenant à l'arbre couvrant. L'argumentaire déployé tient aussi pour leur algorithme probabiliste d'élection. Le principe de la construction de l'arbre est le suivant : le site élu devient la racine et propage un jeton dans le réseau en choisissant un de ses voisins aléatoirement. A la première réception du jeton, chaque site marque l'émetteur comme son père puis réexpédie ce jeton à l'un de ses voisins. Le jeton circule donc comme une marche aléatoire. En un temps fini, tous les sites sont atteints et possèdent donc un père. La construction de l'arbre est alors terminée.

Dans [IJ90], les auteurs proposent un algorithme auto-stabilisant d'exclusion mutuelle écrit dans le modèle à états. Cet algorithme est basé sur la circulation aléatoire d'un jeton qui symbolise le privilège d'accéder à la section critique. L'algorithme est rendu auto-stabilisant : si plusieurs jetons sont présents sur le réseau, les auteurs proposent de les fusionner, dés lors que ceux-ci sont présents sur le même site. Grâce à la propriété de rencontre des marches aléatoires, l'algorithme converge donc en temps fini vers une configuration où il n'existe qu'un seul jeton. Dans [IKOY02], les auteurs améliorent l'équité de l'algorithme (c'est-à-dire que les chances d'obtenir le jeton sont égales) en assignant des pondérations à chacune des arêtes du graphe de communication.

Enfin dans [DSW02], partant du constat que si les réseaux possèdent un degré de dynamisme trop important, il n'existe pas de solution satisfiable en terme de complexité, les auteurs proposent d'utiliser des paradigmes différents pour concevoir des algorithmes distribués, de manière à simplifier au maximum l'écriture de ceux-ci. Ils proposent dans le cadre des réseaux ad-hoc, une solution auto-stabilisante pour la communication de groupe en utilisant les marches aléatoires. Les algorithmes basés sur des marches aléatoires permettent d'éviter au maximum d'avoir à gérer le dynamisme des réseaux tant que le degré de dynamisme n'altère pas les propriétés de percussion, de couverture et de rencontre.

2.6 Conclusion

Après avoir rappeler le contexte de nos travaux, nous avons présenté les deux outils que nous combinons pour concevoir des algorithmes de contrôle distribué. Par leurs propriétés, les marches aléatoires sont un paradigme viable en algorithmique distribuée : elles permettent la couverture d'un graphe de manière rapide, certes moins qu'un algorithme déterministe, mais cela entraîne d'une part une simplification non négligeable de l'écriture des algorithmes, et d'autre part de s'affranchir de la gestion du dynamisme des réseaux. Le mot circulant permet lui une "centralisation mobile" de la collecte et la diffusion d'informations. Nous nous servons de ce dernier dans l'écriture d'algorithmes auto-stabilisants, pour proposer une solution alternative et moins coûteuse que les procédures d'inondation mises en oeuvre dans certains réseaux dynamiques.

Chapitre 3

Automatisation du calcul de valeurs caractéristiques d'algorithmes distribués fondés sur des marches aléatoires

Résumé : *Dans ce chapitre, nous présentons les résultats que nous avons obtenus sur l'évaluation de la complexité d'algorithmes distribués à base de marches aléatoires. Nous utilisons l'analogie entre le comportement d'une marche aléatoire sur un graphe et le comportement du réseau électrique associé. La plupart des résultats connus font appel à la notion de résistance. Notre algorithme permet le calcul automatique des temps de percussion, de commutation et de couverture qui peuvent s'exprimer directement à l'aide de résistances. Ces résultats ont fait l'objet de deux publications [BBBS03] et [BBSB04].*

3.1 Introduction

Comme nous l'avons vu (Section 2.5), l'évaluation de la complexité en messages d'algorithmes distribués fondés sur des marches aléatoires peut se faire directement à travers les résultats de la théorie des chaînes de Markov. Elle peut aussi s'effectuer via la théorie du potentiel.

Nous avons choisi cette deuxième approche qui consiste ([DS00]) à mener une analyse parallèle entre le comportement d'une marche aléatoire et celui d'un réseau électrique. Le calcul des différentes caractéristiques d'une marche aléatoire est en outre moins coûteux en utilisant l'approche que nous développons ici.

En effet, les algorithmes basés sur des marches aléatoires utilisent en général les propriétés de *percussion*, de *couverture* et de *rencontre* (cf. Section 2.5.2) pour mener à bien une tâche. Nous allons donc fournir une méthode qui fournira les évaluations des temps de percussion et de couverture, à partir de la topologie du réseau d'interconnexion.

Notre méthode utilise la relation entre résistances électriques et marches aléatoires établie dans [DS00, CRR+97]. Nous fournissons une méthode automatique pour calculer les résistances sur un graphe modélisant le système distribué.

Nous commençons par rappeler l'analogie qu'il existe entre marches aléatoires et réseaux électriques. Puis nous rappelons les valeurs des temps de percussion et de couverture en fonction des différentes résistances du graphe. Ensuite nous exposons notre méthode à l'aide d'un exemple détaillé, puis nous présentons l'algorithme permettant le calcul automatique des résistances dans un réseau électrique. Nous détaillons les résultats que fournit notre méthode en particulier sur l'anneau et le graphe complet. Nous terminons en montrant comment calculer le temps de rencontre entre plusieurs marches aléatoires en utilisant cette méthode.

3.2 Réseaux de résistances électriques

Le calcul des grandeurs physiques du réseau résistif associé permet l'évaluation de certains résultats sur les marches aléatoires.

Observation de la marche aléatoire

Soit le graphe suivant :

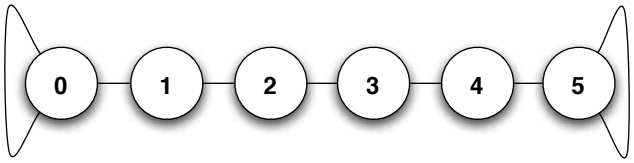


FIG. 3.1 – Graphe G

Considèrons une marche aléatoire sur le graphe G. Soit $P(x)$ la probabilité, partant de x, que l'événement "être sur 5" se produise avant l'événement "être sur 0". L'observation de $P(x)$ comme une fonction définie sur les points $x = 0, 1, \ldots, 5$ fournit les propriétés suivantes :

1. $P(0) = 0$
2. $P(5) = 1$
3. $P(x) = \frac{1}{2}P(x-1) + \frac{1}{2}P(x+1), \forall x \in \{1, 2, \ldots 4\}$

Les deux premières propriétés formalisent la convention (0 et 5 sont des puits, 0 est l'état perdant et 5 l'état gagnant). La dernière propriété vient d'un résultat de probabilité :

Théorème 3.1 (Bayes) *Soit A, un évènement, et E l'ensemble des évenements (A excepté), alors*

$$P(A) = \sum_{i=1}^{n} (P(A|E_i)) \times P(E_i)$$

Donc pour un sommet i seuls les sommets $i+1$ et $i-1$ fournissent des probabilités non nulles (par exemple $P(4|1) = 0$) et ces probabilités sont de $\frac{1}{2}$ (une chance sur deux d'aller à droite ou bien à gauche).

Observation du potentiel dans le réseau électrique associé

Soit le réseau résistif suivant :

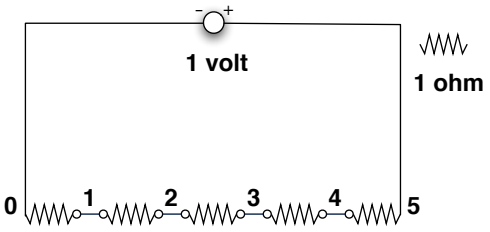


FIG. 3.2 – Réseau résistif associé au graphe G

Le potentiel en chacun des points est donné par les mêmes lois que $P(x)$, à savoir :
- $v(0) = 0$
- $v(5) = 1$
- $v(x) = \frac{v(x-1)+v(x+1)}{2}$

Les deux premières propriétés proviennent de l'application du potentiel. La troisième propriété de l'application de la loi de Kirchoff et du théorème de Millman.

3.3 Rappels d'électrocinétique

Nous rappelons ici quelques bases de l'électrocinétique, qui nous permettent dans la suite de nos travaux une évaluation automatique des complexités en moyenne d'algorithmes fondés sur des marches aléatoires.

Le réseau électrique associé au graphe est construit en plaçant sur chaque arête une résistance unitaire.

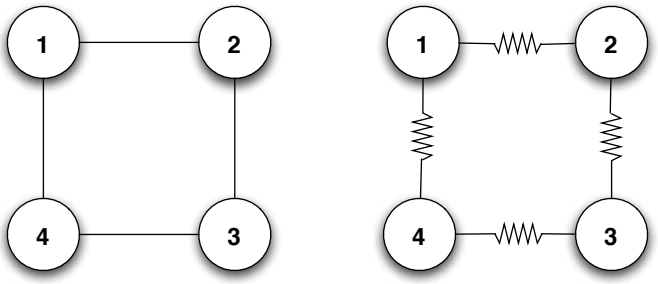


FIG. 3.3 – Construction du réseau de résistances

Théorème 3.2 (Loi d'Ohm) *Il existe une relation entre l'intensité, la résistance et le potentiel entre deux noeuds A et B :*

$$u_{AB} = r_{AB} \times i_{AB}$$

r_{AB} est la résistance du dipôle placé entre A et B.

Théorème 3.3 (Association de résistances) Association en série *Etant données deux résistances (réseaux électriques) de valeurs r_1 et r_2 associées en série, la valeur de la résistance équivalente r_{eq} est définie par :*

$$r_{eq} = r_1 + r_2$$

Association en parallèle *Etant données deux résistances (réseaux électriques) de valeurs r_1 et r_2 associées en parallèle, la valeur de la résistance équivalente r_{eq} est définie par :*

$$\frac{1}{r_{eq}} = \frac{1}{r_1} + \frac{1}{r_2}$$

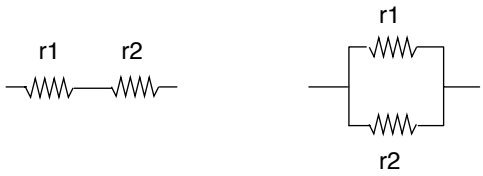


FIG. 3.4 – Association de résistances série/parallèle

Théorème 3.4 (Transformation étoile triangle) *Si un noeud v possède trois voisins a, b, c avec $r_{av} = A, r_{bv} = B, r_{cv} = C$ (cf. Fig 3.5), il est possible de réécrire la portion en étoile du réseau en triangle avec*

$$r_{ab} = \frac{AB + BC + CA}{C}, r_{ac} = \frac{AB + BC + CA}{B}, r_{bc} = \frac{AB + BC + CA}{A}$$

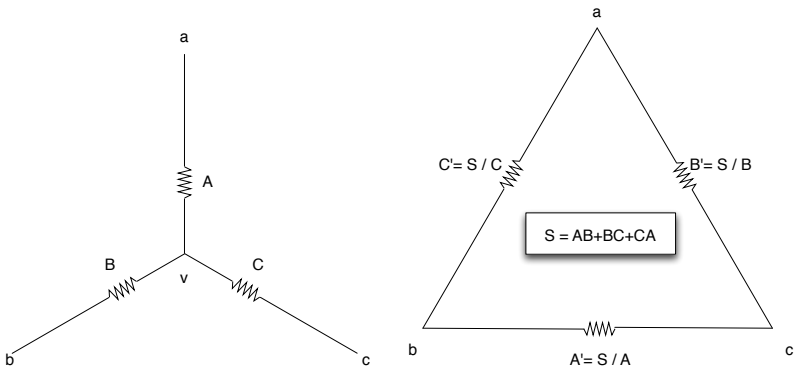


FIG. 3.5 – Transformation triangle étoile

Théorème 3.5 (Loi de Kirchhoff) *En chaque nœud du réseau électrique, la somme des courants entrants est égale à la somme des courants sortants.*

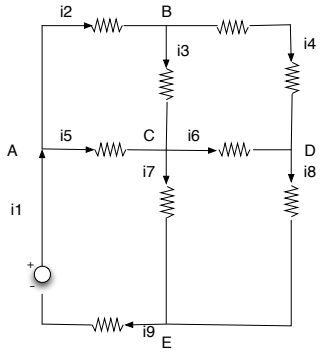


FIG. 3.6 – Schéma de la loi de Kirchoff

Cette loi suppose que les intensités sont déjà orientées (l'orientation se fait en potentiel décroissant). Ici :

- en B $i2 = i4 + i3$
- en C $i5 + i3 = i6 + i7$
- ...

En injectant la loi d'Ohm sur l'écriture de la loi de Kirchoff, il est facile d'obtenir le théorème de Millman qui permet de calculer tous types de grandeurs dans un réseau électrique.

Théorème 3.6 (Théorème de Millman)

$$\frac{\frac{V-V_1}{r_1} + \frac{V-V_2}{r_2} + \cdots + \frac{V-V_n}{r_n}}{\frac{1}{r_1} + \frac{1}{r_2} + \cdots + \frac{1}{r_n}} = 0$$

Ce théorème est issu de la loi de Kirchoff et de la loi d'Ohm, dans notre cas (résistances unitaires), il est possible de transformer cette relation ($\forall i, r_i = 1$). Ici :

$$nV - V_1 - V_2 - \cdots - V_n = 0$$

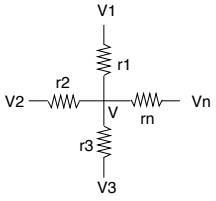


FIG. 3.7 – Schéma de Millman

3.4 Résultats à partir de l'analogie avec les réseaux électriques

Les résultats fournis ici viennent de [CRR+97]. Ces résultats permettent d'analyser différemment le comportement d'une marche aléatoire sur un graphe, et donc permettent un calcul de la complexité en messages des algorithmes distribués fondés sur des marches aléatoires.

Définition 3.1 *La résistance entre 2 points A et B (notée $R(A, B)$) est définie comme étant la résistance globale de tout le réseau électrique entre ces deux points.*

Théorème 3.7 *Pour 2 sommets u et v dans G, le temps de commutation est :*

$$C(u, v) = 2 \times m \times R(u, v)$$

Définition 3.2 *Résistance d'un graphe*
La résistance R d'un graphe est la résistance maximum effective entre deux sommets du graphe.

$$R = \max_{(i,j)\in V^2} R(i, j)$$

Théorème 3.8 *Le temps de couverture d'un graphe G de n sommets et m arêtes est encadré par :*

$$m \times R \leq C \leq O(m \times R \times \log n)$$

Le principe de Rayleigh [Ray99]) permet de mieux appréhender la valeur de R en modifiant la valeur d'une résistance.

Principe 3.1 (Rayleigh's "Short/Cut") *La résistance n'est jamais augmentée en diminuant la résistance d'une arête (en court-circuitant deux nœuds) et n'est jamais diminuée en augmentant la résistance d'une arête (en la supprimant). De manière similaire la résistance n'est jamais diminuée en supprimant un noeud, laissant chaque arête adjacente attachée à une des moitiés du noeud.*

L'ajout d'un lien permet alors globalement de réduire R et la suppression de ce lien d'augmenter R.

A partir d'une comparaison entre R et la résistance R_{span} d'un arbre couvrant du graphe, les auteurs de [CRR+97] déduisent que :

Corollaire 3.1

$$C \leq 2mR_{span}$$

R_{span} étant défini comme le minimum, parmi tous les arbres couvrants T, de la somme des résistances effectives des arêtes de T.

[Tet91] fournit une évaluation du temps de percussion entre deux sommets à partir de la résistance entre ces deux sommets dans le réseau électrique associé :

Théorème 3.9

$$h(i,j) = mR(i,j) + \frac{1}{2}\sum_{k\in V} \deg(k) \times [R(j,k) - R(i,k)]$$

Ce résultat a été généralisé aux cas des graphes valués dans [BS04].

Théorème 3.10

$$h(i,j) = \omega(G)R(i,j) + \frac{1}{2}\sum_{k\in V} \omega(k) \times [R(j,k) - R(i,k)]$$

avec $\omega(G) = 1/2 \times \sum_{(i,j)\in V^2} \omega(i,j)$ *et* $\omega(k) = \sum_{j\in V ois_k} \omega(j,k)$, *et* $\omega(j,k)$ *est le poids de l'arête* (j,k)

Dans [Fei95a, Fei95b], l'auteur utilise le Théorème 3.8 pour fournir des bornes au temps de couverture :

Théorème 3.11

$$(1 + o(1))n \ln l < C < \frac{4}{27}n^3 + o(n^3)$$

Nous reformulons le temps de couverture cyclique en fonction des résistances du graphe :

Définition 3.3 *Pour un graphe symétrique, le temps de couverture cyclique est défini par :*

$$CCT = \min\left\{\sum_{j=1}^{n-1} m \times R(\sigma(j), \sigma(j+1)) | \sigma \in \mathcal{S}_n\right\}$$

où $\mathcal{S}_n$ *est l'ensemble des permutations sur* $\{1, \cdots, n\}$

Toutes ces notions permettent le calcul des complexités en moyenne des algorithmes distribués à base de marches aléatoires. Par exemple l'algorithme introduit dans [BIZ89] calcule un arbre couvrant une fois que tous les sites du réseaux sont visités. Nous pouvons à l'aide de l'encadrement fourni précédemment et de la méthode que nous développons ici, fournir des bornes à la complexité de cet algorithme.

Dans [DSW02], les auteurs initialisent les valeurs des horloges de garde en fonction du temps de couverture, ce qui leur permet grâce à un paramètre de sécurité de garantir l'auto-stabilisation de leur algorithme.

Nous n'avons pas détaillé le calcul des temps de rencontre entre plusieurs marches aléatoires. Nous abordons, en dernière section, une manière de le calculer pour deux marches à partir du calcul de temps de percussion. Cette méthode est généralisable au cas de n marches aléatoires sur le même graphe.

3.5 Méthode générale

Pour calculer la résistance globale entre deux sommets d'un réseau électrique, deux approches sont possibles :

- La transformation des réseaux (calcul des résistances équivalentes sur des résistances associées en parallèle ou en série, ou la transformation triangle-étoile cf. Théorèmes 3.3 et 3.4). Néanmoins, cette approche n'est pas satisfaisante dans le sens où elle ne permet pas :
  - de calculer une résistance globale dans des cycles avec des fuites de courant (les suppressions des cycles recréent des nouveaux cycles).
  - d'évaluer systématiquement les résistances.
- Le calcul automatique (à l'aide du théorème de Millman cf. Théorème 3.6) Le principe global utilisé dans la méthode décrite ici, est la loi d'Ohm (cf. Théorème 3.2). L'idée est de fixer une différence de potentiel entre les deux points concernés et de pouvoir en déduire des propriétés locales qui permettront le calcul de l'intensité globale passant à travers l'ensemble du circuit.

Exemple de calcul de résistance

Soit le graphe suivant :

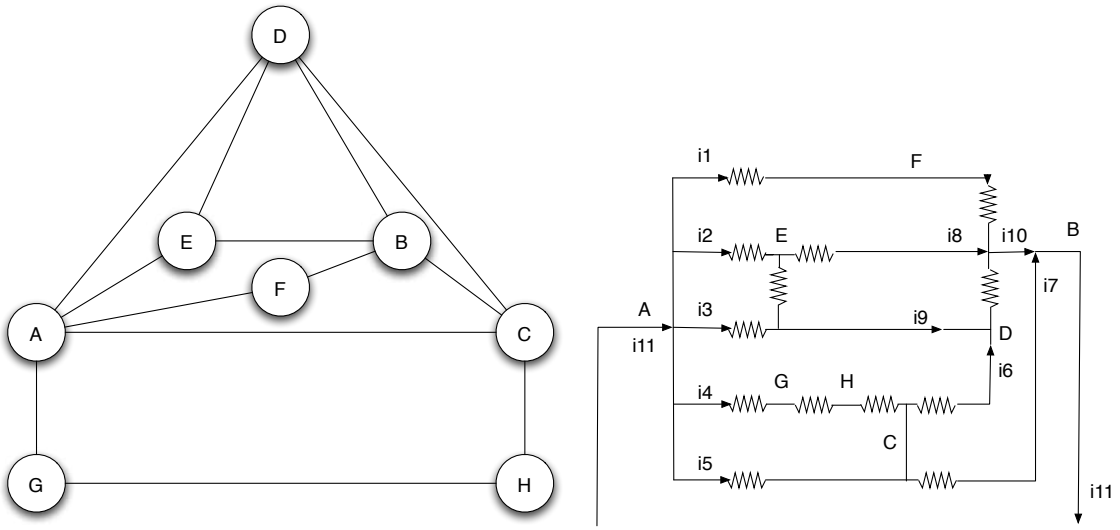


FIG. 3.8 – Graphe et réseau résistif

En appliquant une différence de potentiel de 1 volt entre A et B ($V_A = 1$ volt, le maximum, remarque importante pour les calculs et $V_B = 0$ volt), il est alors possible d'écrire le théorème de Millman sur tous les sommets (exeptés A et B, dont les potentiels sont déjà connus). Il en résulte alors un système de $n-2$ équations à $n-2$ inconnues

qu'il est possible de résoudre. Ici,

$$\begin{cases} -V_A & -V_B & & & & +2V_F & & & = & 0 \\ -V_A & -V_B & & -V_D & +3V_E & & & & = & 0 \\ -V_A & -V_B & -V_C & +4V_D & -V_E & & & & = & 0 \\ -V_A & & & & & & +2V_G & -V_H & = & 0 \\ & & -V_C & & & & -V_G & +2V_H & = & 0 \\ -V_A & -V_B & +4V_C & -V_D & & & & -V_H & = & 0 \end{cases}$$

V_A et V_B sont connus, en les remplacant :

$$\iff \begin{cases} & & & +2V_F & & & = & 1 \\ & -V_D & +3V_E & & & & = & 1 \\ -V_C & +4V_D & -V_E & & & & = & 1 \\ & & & & +2V_G & -V_H & = & 1 \\ -V_C & & & & -V_G & +2V_H & = & 0 \\ 4V_C & -V_D & & & & -V_H & = & 1 \end{cases}$$

Après résolution du système :

$$\begin{cases} V_A & = & 1 \\ V_B & = & 0 \\ V_C & = & \frac{11}{20} \\ V_D & = & \frac{1}{2} \\ V_E & = & \frac{1}{2} \\ V_F & = & \frac{1}{2} \\ V_G & = & \frac{17}{20} \\ V_H & = & \frac{14}{20} \end{cases}$$

Il est donc possible de calculer, grâce à la loi d'Ohm et à la loi de Kirchoff, les intensités (orientées avant grâce aux valeurs des potentiels : une intensité va dans le sens décroissant du potentiel) dans chacune des arêtes.

$$\begin{cases} i_1 & = & \frac{1-\frac{1}{2}}{1} & = & \frac{1}{2} \\ i_2 & = & \frac{1-\frac{1}{2}}{1} & = & \frac{1}{2} \\ i_3 & = & \frac{1-\frac{1}{2}}{1} & = & \frac{1}{2} \\ i_4 & = & \frac{1-\frac{17}{20}}{1} & = & \frac{3}{20} \\ i_5 & = & \frac{1-\frac{11}{20}}{1} & = & \frac{9}{20} \\ i_6 & = & \frac{\frac{11}{20}-\frac{1}{2}}{1} & = & \frac{1}{20} \\ i_7 & = & \frac{\frac{11}{20}-0}{1} & = & \frac{11}{20} \\ i_8 & = & \frac{\frac{1}{2}-0}{1} & = & \frac{1}{2} \\ i_9 & = & i_2+i_3-i_8 & = & \frac{1}{2} \\ i_{10} & = & i_1+i_8+i_9+i_6 & = & \frac{31}{20} \\ i_{11} & = & i_{10}+i_7 & = & \frac{21}{10} \end{cases}$$

Nous pouvons donc calculer l'intensité des courants qui parcourent le circuit :

$$i_{entree} = i_{11} = i_1 + i_2 + i_3 + i_4 + i_5 = \frac{42}{20}$$

Donc en appliquant la loi d'Ohm :

$$R = \frac{U}{I} = \frac{1}{\frac{21}{10}} = \frac{10}{21} \text{ ohms}$$

3.6 Algorithme de calcul

Pour calculer la résistance effective entre deux sommets ($R(h,k)$), l'algorithme applique les différentes étapes que nous avons décrites dans la section précédente.

3.6.1 Architecture de l'algorithme

Algorithme 2 Calcul de $R(h,k)$

1: Construire la matrice de Millman (étape 1).
2: Placer les potentiels $V_h = 1$ et $V_k = 0$ (étape 2).
3: Résoudre le système linéaire (étape 3).
4: Calcul du courant sortant en k (étape 4).
5: Application globale de la loi d'Ohm sur le circuit $U = R \times I$ (étape 5).

Dans la suite nous détaillons les fonctions principales de l'algorithme.

3.6.2 Construction de la matrice de Millman (1)

Cette partie de l'algorithme détermine le système d'équations linéaires provenant de l'application du théorème de Millman sur chacun des sommets du graphe.

1: **pour tout** $i \in V \backslash \{h;k\}$ **faire**
2: **pour tout** $j \in V$ **faire**
3: **si** $i = j$ **alors**
4: $M_{ij} \longleftarrow -\#(Vois_i)$
5: **fin si**
6: **si** $j \in Vois_i$ **alors**
7: $M_{ij} \longleftarrow 1$
8: **sinon**
9: $M_{ij} \longleftarrow 0$
10: **fin si**
11: **fin pour**
12: **fin pour**

3.6.3 Placer les potentiels (2)

Nous complétons la matrice pour prendre en compte l'application des potentiels au noeud x. Nous obtenons $M = (M_{ij})_{(i,j)\in V\times V}$

1: **pour tout** $i \in V$ **faire**

2: **si** $i = x$ **alors**
3: $M_{xi} \longleftarrow 1$
4: **sinon**
5: $M_{xi} \longleftarrow 0$
6: **fin si**
7: **fin pour**

3.6.4 Résoudre le système linéaire (3)

Cette opération consiste à obtenir les potentiels de chaque noeud, en résolvant $M \times V = S$ où S est le vecteur avec toutes les entrées égales à 0 sauf pour la h-ième ligne où elle sera égale à 1.

3.6.5 Calcul du courant sortant en k (4)

Connaissant les potentiels sur chaque noeud, il devient possible de calculer grâce à la loi d'Ohm, le courant sortant de k :

1: $current \longleftarrow 0$
2: **pour tout** $j \in Vois_h$ **faire**
3: $current \longleftarrow current + V_k - V_j$ /* *Le dipôle résistif entre* i *et* j *vaut* $r(i,j) = 1\Omega$ */
4: **fin pour**

3.6.6 Application globale de la loi d'Ohm (5)

L'intensité de courant circulant entre h et k est connue, la différence de potentiel aussi. Pour obtenir la résistance effective entre h et k, il suffit d'appliquer la loi d'Ohm :

1: Retourner $1/courant$ /* $R(h,k) = (V_h - V_k)/courant$ *avec* $V_h - V_k = 1$ *)* */

3.7 Exemples

3.7.1 Un exemple d'application sur un graphe arbitraire

Soit G le graphe présenté Figure 3.8 et le réseau résistif qui lui est associé, la matrice des résistances suivante est obtenue par l'application de l'algorithme décrit Section 3.6 sur chaque paire de sommets

$$\begin{pmatrix} 0 & \frac{202}{419} & \frac{177}{419} & \frac{172}{419} & \frac{203}{419} & \frac{260}{419} & \frac{299}{419} & \frac{358}{419} \\ \frac{202}{419} & 0 & \frac{199}{419} & \frac{178}{419} & \frac{205}{419} & \frac{260}{419} & \frac{441}{419} & \frac{440}{419} \\ \frac{177}{419} & \frac{199}{419} & 0 & \frac{187}{419} & \frac{266}{419} & \frac{347}{419} & \frac{358}{419} & \frac{299}{419} \\ \frac{172}{419} & \frac{178}{419} & \frac{187}{419} & 0 & \frac{195}{419} & \frac{334}{419} & \frac{417}{419} & \frac{422}{419} \\ \frac{203}{419} & \frac{205}{419} & \frac{266}{419} & \frac{195}{419} & 0 & \frac{363}{419} & \frac{464}{419} & \frac{485}{419} \\ \frac{260}{419} & \frac{260}{419} & \frac{347}{419} & \frac{334}{419} & \frac{363}{419} & 0 & \frac{529}{419} & \frac{558}{419} \\ \frac{299}{419} & \frac{441}{419} & \frac{358}{419} & \frac{417}{419} & \frac{464}{419} & \frac{529}{419} & 0 & \frac{299}{419} \\ \frac{358}{419} & \frac{440}{419} & \frac{299}{419} & \frac{422}{419} & \frac{485}{419} & \frac{558}{419} & \frac{299}{419} & 0 \end{pmatrix}$$

FIG. 3.9 – Matrice des résistances associée au graphe G

Une fois toutes les résistances connues, les différents temps peuvent être aisément calculés :

– La matrice des temps de percussion :

$$\begin{pmatrix} 0 & \frac{3010}{419} & \frac{2595}{419} & \frac{2538}{419} & \frac{3467}{419} & \frac{5220}{419} & \frac{6267}{419} & \frac{7132}{419} \\ \frac{2242}{419} & 0 & \frac{2497}{419} & \frac{2232}{419} & \frac{3109}{419} & \frac{4836}{419} & \frac{7729}{419} & \frac{7814}{419} \\ \frac{2007}{419} & \frac{2677}{419} & 0 & \frac{2439}{419} & \frac{3992}{419} & \frac{6057}{419} & \frac{6740}{419} & \frac{6071}{419} \\ \frac{1934}{419} & \frac{2396}{419} & \frac{2423}{419} & 0 & \frac{3061}{419} & \frac{5880}{419} & \frac{7499}{419} & \frac{7662}{419} \\ \frac{1811}{419} & \frac{2221}{419} & \frac{2924}{419} & \frac{2009}{419} & 0 & \frac{5731}{419} & \frac{7584}{419} & \frac{7955}{419} \\ \frac{1540}{419} & \frac{1924}{419} & \frac{2965}{419} & \frac{2804}{419} & \frac{3707}{419} & 0 & \frac{7417}{419} & \frac{7892}{419} \\ \frac{1507}{419} & \frac{3737}{419} & \frac{2568}{419} & \frac{3343}{419} & \frac{4480}{419} & \frac{6337}{419} & 0 & \frac{3985}{419} \\ \frac{2176}{419} & \frac{3626}{419} & \frac{1703}{419} & \frac{3310}{419} & \frac{4655}{419} & \frac{6616}{419} & \frac{3789}{419} & 0 \end{pmatrix}$$

FIG. 3.10 – Matrice des temps de percussions associée au graphe G

– Limite au temps de couverture :

$$13 \times \frac{558}{419} < C < (2 \times 13 \times \frac{558}{419} \times (1 + \log(8))$$

– La matrice des temps de commutation :

$$\begin{pmatrix} 0 & \frac{5252}{419} & \frac{4602}{419} & \frac{4472}{419} & \frac{5278}{419} & \frac{6760}{419} & \frac{7774}{419} & \frac{9308}{419} \\ \frac{5252}{419} & 0 & \frac{5174}{419} & \frac{4628}{419} & \frac{5330}{419} & \frac{6760}{419} & \frac{11466}{419} & \frac{11440}{419} \\ \frac{4602}{419} & \frac{5174}{419} & 0 & \frac{4862}{419} & \frac{6916}{419} & \frac{9022}{419} & \frac{9308}{419} & \frac{7774}{419} \\ \frac{4472}{419} & \frac{4628}{419} & \frac{4862}{419} & 0 & \frac{5070}{419} & \frac{8684}{419} & \frac{10842}{419} & \frac{10972}{419} \\ \frac{5278}{419} & \frac{5330}{419} & \frac{6916}{419} & \frac{5070}{419} & 0 & \frac{9438}{419} & \frac{12064}{419} & \frac{12610}{419} \\ \frac{6760}{419} & \frac{6760}{419} & \frac{9022}{419} & \frac{8684}{419} & \frac{9438}{419} & 0 & \frac{13754}{419} & \frac{14508}{419} \\ \frac{7774}{419} & \frac{11466}{419} & \frac{9308}{419} & \frac{10842}{419} & \frac{12064}{419} & \frac{13754}{419} & 0 & \frac{7774}{419} \\ \frac{9308}{419} & \frac{11440}{419} & \frac{7774}{419} & \frac{10972}{419} & \frac{12610}{419} & \frac{14508}{419} & \frac{7774}{419} & 0 \end{pmatrix}$$

FIG. 3.11 – Matrice des temps de commutation associée au graphe G

– Le temps de couverture cyclique

$$CCT = \min_{\sigma \in S_n} \left(\sum_{i \in \{1,\cdots,n-1\}} m \times R(\sigma(i), \sigma(i+1)) \right) = \frac{12856}{419}$$

3.7.2 Etude analytique sur l'anneau et le graphe complet

Nous montrons ici deux cas où l'utilisation des lois de l'électricité aboutit à des résultats généraux sur les temps de percussion et de couverture.

Cas de l'anneau

L'application des lois fondamentales de l'électricité (approche du physicien) peut parfois permettre un calcul rapide de la valeur maximale des résistances et fournit donc des résultats sur les différents temps de calcul. Par exemple, sur un anneau, grâce aux propriétés d'association de résistances, la résistance effective entre deux noeuds i et j, est :

$$R(i,j) = \frac{-l^2 + l \times n}{n}$$

où l est la distance du plus court chemin entre i et j et la résistance maximale effective est :

$$R_{Max} = \begin{cases} \frac{n}{4} & \text{si } n \text{ est pair} \\ \frac{n^2-1}{4n} & \text{si } n \text{ est impair} \end{cases}$$

Il est donc aisé d'en déduire les valeurs suivantes :

– Temps de commutation :

$$C(i,j) = 2mR(i,j) = 2n \times \frac{-l^2 + l \times n}{n} = 2(l(n-l))$$

où l est la distance du plus court chemin entre i et j

– Temps de percussion (obtenue grâce à la symétrie du graphe) :

$$h(i,j) = l(n-l)$$

– Bornes au temps de couverture :
  – si n est pair :
  $$\frac{n^2}{4} < C < O\left(\frac{n^2}{4} \times \log(n)\right)$$
  – si n est impair :
  $$\frac{n^2-1}{4} < C < O\left(\frac{n^2-1}{4} \times \log(n)\right)$$

– Temps de couverture cyclique : la résistance minimum est obtenue entre deux sommets voisins dans l'anneau, elle vaut $\frac{n-1}{n}$. Puisque tous les autres termes sont constants, le calcul du temps de couverture cyclique est donc obtenu en minimisant le terme résistif.

$$CCT = \min\left\{\sum_{j=1}^{n-1} m \times R\left(\sigma(j), \sigma(j+1)\right) \backslash \sigma \in \mathcal{S}_n\right\}$$
$$= \sum_{j=1}^{n-1} n\left(\frac{n-1}{n}\right) = (n-1)^2$$

Cas du graphe complet

En appliquant notre algorithme au cas du graphe complet, nous avons remarqué que toutes les résistances étaient égales (effectivement, le graphe est régulier). Elles sont égales à $\frac{2}{n}$.

Proposition 3.1 *Dans le cas du graphe complet $G = (V, E)$, toutes les résistances (sauf le cas trivial $R(i,i) = 0, \forall i \in V$) sont égales à $\frac{2}{n}$ où n est le nombre de sommets de G*

Démonstration Calculons la résistance effective entre 0 et 1 en plaçant les potentiels V_0 et V_1 respectivement à 0 volt et 1 volt. Grâce à la symétrie du graphe $\forall (i,j) \in \{2; \ldots; n-1\}^2, V_i = V_j$: il n'y a pas de courant qui circule dans les arêtes (i,j). Ainsi le courant sortant de 0 est :

$$\sum_{i=1}^{n-1} I(i,0) = \sum_{i=1}^{n-1} r(0,i)\left(V_i - V_0\right)$$
$$= -\sum_{i=1}^{n-1} V_i \text{ comme } r(0,i) = 1 \text{ et } V_0 = 0$$
$$= -(n-2)V_2 - V_1$$

Le courant sortant de 1 est :

$$\sum_{i=2}^{n-1} I(1,i) + I(1,0) = \sum_{i=1}^{n-1} r(i,1)\left(V_i - V_1\right) + r(0,1)\left(V_0 - V_1\right)$$
$$= \sum_{i=2}^{n-1} V_i - (n-1)V_1 \text{ comme } r(1,i) = 1 \text{ et } V_0 = 0,\ V_1 = 1$$
$$= (n-2)V_2 - (n-1)V_1$$

Comme le courant est injecté seulement en 1 et sort uniquement en 0, ces deux courants sont égaux, et :

$$2(n-1)V_2 = (n-2)V_1$$

Ainsi,

$$V_2 = \frac{1}{2}$$

Alors, le courant sortant en 1 est $\frac{n-2}{2} - (n-1) = -\frac{n}{2}$. Donc,

$$R(0,1) = \frac{V_0 - V_1}{-\frac{n}{2}} = \frac{2}{n}$$

□

Nous sommes maintenant en mesure de fournir des résultats généraux sur les graphes complets :

– Le temps de commutation entre deux noeuds i et j :

$$2mR(i,j) = 2 \times \frac{n \times (n-1)}{2} \times \frac{2}{n} = 2(n-1)$$

– Bornes au temps de couverture :

$$\begin{aligned} mR_{max} <& C < O\left(mR_{max}\log(n)\right) \\ n-1 <& C < O\left((n-1)\log(n)\right) \end{aligned}$$

– Le temps de percussion entre deux noeuds est :

$$\begin{aligned} h(i,j) &= mR(i,j) + \frac{1}{2}\sum_{w\in V} d(w)[R(w,j) - R(w,i)] \\ &= mR(i,j) = \frac{n \times n-1}{2} \times \frac{2}{n} = n-1 \end{aligned}$$

– Enfin le temps de couverture cyclique :

$$\begin{aligned} CCT &= \sum_{i\in\{1,\cdots,n-1\}} m \times R(\sigma(i),\sigma(i+1)) \\ &= \sum_{i\in\{1,\cdots,n-1\}} n-1 = (n-1)^2 \end{aligned}$$

3.8 Calcul du temps de rencontre

Il est possible de modéliser plusieurs marches aléatoires sur un graphe G comme une marche aléatoire sur un autre graphe ($\mathcal{G}$) calculé à partir de G et de l'ordonnancement des déplacements des différentes marches (cf. Section 2.5.4). Nous proposons ici un exemple de modélisation de deux marches aléatoires sur le graphe G ordonnancées de manière asynchrone (et avec un ordonnancement équitable entre les marches aléatoires), en une marche aléatoire sur le graphe $\mathcal{G}$.

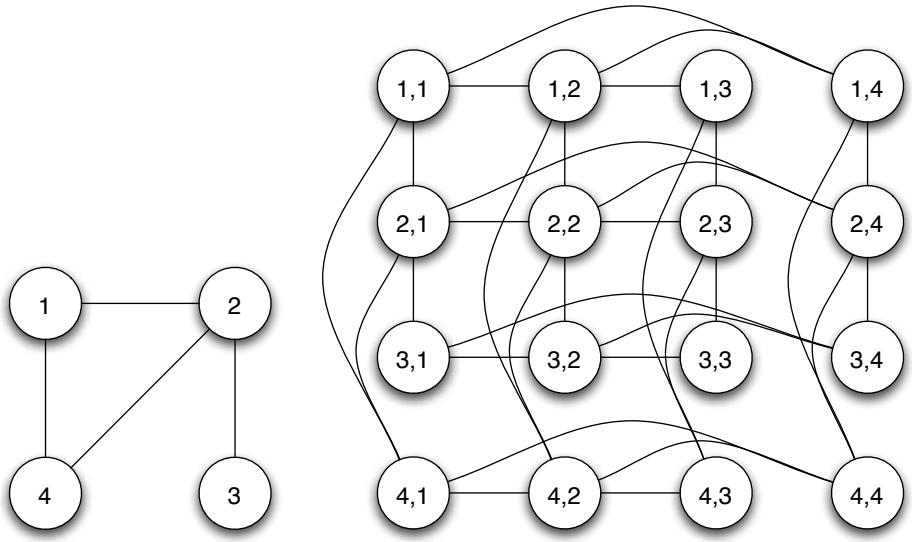


FIG. 3.12 – Graphe G et graphe $\mathcal{G}$ produit avec un ordonnancement asynchrone

Il est alors possible à partir du calcul des résistances dans le graphe $\mathcal{G}$ de calculer les valeurs des temps de percussion. Or les temps de percussions d'un sommet (i, j) avec $i \neq j$ de $\mathcal{G}$ vers les sommets du type (h, h), sont, dans le graphe G, les temps de rencontre sur le sommet h entre deux marches partant de i et de j.

Donc dès lors que $\mathcal{G}$ est calculé, le calcul du temps de rencontre entre deux marches aléatoires se résume à un simple calcul de temps de percussion. Nous fournissons ici l'algorithme permettant le calcul de $\mathcal{G}$ dans le cadre de deux marches aléatoires ordonnées de manière synchrone.

Algorithme 3 Calcul de $\mathcal{G}$ à partir de G pour deux marches aléatoires dans un système asynchrone

Soient $G = (V, E)$ et $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ avec $|\mathcal{V}| = n^2$, les sommets de $\mathcal{V}$ étant numérotés de la forme (i, j) pour i et j dans V
Nous cherchons maintenant à calculer $\mathcal{E}$ l'ensemble des arêtes de $\mathcal{G}$
pour $i \in V$ **faire**
  pour $j \in V$ **faire**
    pour $h \in V$ **faire**
      si $(i, h) \in E$ **alors**
        $\mathcal{E} \leftarrow \mathcal{E} \cup \{((i, j), (h, j))\}$
      fin si
      si $(j, h) \in V$ **alors**
        $\mathcal{E} \leftarrow \mathcal{E} \cup \{((i, j), (i, h))\}$
      fin si
    fin pour
  fin pour
fin pour

Pour un calcul de temps de rencontre entre k marches aléatoires, le graphe $\mathcal{G}$ comportera n^k sommets numérotés comme un vecteur à k composantes, chacune d'elle représentant la position d'une des marches aléatoires. Le cas de l'ordonnancement peut se généraliser facilement, si les marches aléatoires se déplacent simultanément, le calcul de $\mathcal{E}$ devra en tenir compte et considérer k déplacements simultanés.

3.9 Conclusion

Après avoir rappelé et reformulé certaines valeurs caractéristiques des marches aléatoires, nous avons développé une méthode qui permet de fournir un calcul de ces valeurs et donc d'évaluer la complexité des algorithmes distribués basés sur les marches aléatoires.

Cette méthode est illustrée par un exemple, qui met en valeur le caractère automatique de notre calcul de résistances et des différentes valeurs qui en découlent.

Nous avons aussi montré que dans certains cas (exemple : celui de l'anneau) par une simple symétrie et l'application des lois fondamentales de l'électricité, il est possible de donner par une simple formule la valeur de la résistance entre deux points. Dans l'autre cas (celui du graphe complet) l'application de l'algorithme permet de fournir les valeurs des résultats en fonction du nombre de sites qui composent le réseau.

Nous fournissons pour finir une méthode qui permet le calcul des temps de rencontre entre différentes marches aléatoires.

Ces travaux ont été généralisés : dans [BS04], les auteurs proposent un algorithme de calcul exact des temps de percussion sur un graphe valué et dans [BS05], ils proposent un calcul exact du temps de couverture sur un graphe.

Nous avons donc là un outil qui permet de déterminer si les marches aléatoires sont un outil satisfaisant pour la conception d'algorithmes distribués. Outre la simplicité de conception d'algorithmes, les bornes de complexité qu'offrent des marches aléatoires montrent qu'elles sont des solutions satisfaisantes.

Chapitre 4

Mot circulant et maintenance de structures virtuelles

Résumé : *Dans ce chapitre nous utilisons un* mot circulant *pour maintenir une image virtuelle de la topologie du réseau d'interconnexion. Nous avons d'abord introduit dans [BBF04b] un test interne qui permet une correction étape par étape de l'image virtuelle maintenue à travers le mot circulant, et proposé une application : la construction auto-stabilisante d'un arbre couvrant. Nous avons adapté dans [BBFR06a, BBFR06b], la gestion complète du mot circulant, afin de pouvoir maintenir une image virtuelle de la topologie d'un réseau orienté. Cette nouvelle gestion permet la maintenance d'arbres couvrants, mais aussi d'anneaux.*

4.1 Introduction

La maintenance de structures virtuelles est un pré-requis fondamental en algorithmique distribuée : de nombreux algorithmes supposent l'existence d'une structure de communication leur permettant ainsi de se décharger du bon acheminement des messages. Par exemple l'algorithme d'élection de LeLann fonctionne sur un réseau d'interconnexion en forme d'anneau. Si la topologie du réseau est arbitraire, un algorithme construisant et maintenant un anneau doit être préalablement exécuté. Certains autres algorithmes fonctionnent sur d'autres types de topologies : arbres couvrants, grilles, cliques, ...

La structuration d'un réseau est souvent effectuée à l'aide d'algorithmes de diffusion. Ces méthodes permettent d'obtenir rapidement une image de la topologie du réseau. Néanmoins elles peuvent être coûteuses, et nécessitent une ré-exécution en cas de pannes ou de reconfigurations topologiques. D'autres solutions utilisent l'inondation comme paradigme de conception d'algorithmes de structuration. Mais aucun de ces procédés ne permet, quand le dynamisme du réseau est trop important, de garantir l'obtention d'une image valide. Nous proposons donc une solution alternative à l'inondation dont le coût en messages est fixe et dont l'adaptativité reste néanmoins satisfaisante.

Nous avons donc combiné marches aléatoires et mot circulant pour permettre la maintenance d'une structure virtuelle. Le mot circulant est un outil qui permet de collecter de l'information topologique au fur et à mesure de son déplacement au sein du réseau.

Ce mot circule aléatoirement dans le réseau sous forme de jeton, garantissant ainsi la percussion d'un site vers un autre et la couverture du réseau (cf. Section 2.5.2).

Après avoir présenté le mot circulant, nous présentons nos contributions : une version tolérante aux pannes du mot circulant (cf. Section 4.3) et son adaptation aux réseaux orientés (cf. Section 4.4).

4.2 Mot circulant et construction d'arbres

4.2.1 Calcul d'arbres couvrants aléatoires

Broder utilise une marche aléatoire dans [Bro89] pour concevoir un algorithme qui calcule un arbre couvrant uniformément choisi au hasard parmi l'ensemble des arbres couvrants du graphe :

Algorithme 4 Algorithme de Broder

1. Simuler une marche aléatoire sur le graphe G en commençant sur un sommet arbitraire s jusqu'à ce que tous les sommets soient visités. Pour chaque sommet $i \in V - s$, collecter l'arête $\{j, i\}$ qui correspond à la première visite du sommet i. Soit T cet ensemble d'arêtes.
2. Afficher l'ensemble T

Cet algorithme a été conçu pour fonctionner sur un graphe. Dans [BIZ89], Bar-Ilan et Zernick ont proposé d'adapter cet algorithme pour le calcul d'un arbre couvrant aléatoire sur un réseau.

Algorithme 5 Algorithme local au site i de Bar-Ilan et Zernick

$visite \longleftarrow faux$
Réception du jeton J provenant du site j
si $visite = faux$ **alors**
  $visite \longleftarrow vrai$
  $Père \longleftarrow j$
fin si
Envoyer le jeton J à k choisi uniformement au hasard parmi $Vois_i$

Néanmoins cet arbre couvrant, une fois calculé, reste fixe. Il est donc nécessaire après une reconfiguration topologique de vérifier que celui-ci reste "valide". Si ce n'est pas le cas, il faut alors ré-exectuer l'algorithme afin de calculer un nouvel arbre couvrant.

Dans [Fla00], l'auteur présente une adaptation de l'algorithme de Broder qui permet le calcul d'un arbre couvrant adaptatif de manière distribuée. Nous entendons par *arbre couvrant adaptatif*, un arbre en constante évolution. Un tel arbre est résistant aux reconfigurations topologiques : si un canal de communication entre deux sites devient défaillant, ce canal finira par ne plus faire partie non plus de l'arbre couvrant (s'il en faisait partie auparavant). L'algorithme proposé est un algorithme permanent et donc l'occurrence de

reconfigurations topologiques ne nécessite pas d'intervention "extérieure" pour ré-excuter l'algorithme. Il utilise le contenu du mot circulant pour construire un arbre couvrant. En effet, le mot circulant contient l'historique des sites visités par le jeton. Il est alors possible, pour le site détenteur du jeton, de calculer un chemin vers tous les autres sites visités.

Exemple : Le message contenant le mot circulant (nous appelons ce message un *jeton*) circule aléatoirement comme une marche aléatoire (cf. Section 2.5) sur le réseau et effectue la mise à jour du mot qu'il contient. Une fois que tous les sites ont été visités par le jeton, le site possédant le jeton peut alors calculer localement l'image d'un arbre couvrant sur le réseau.

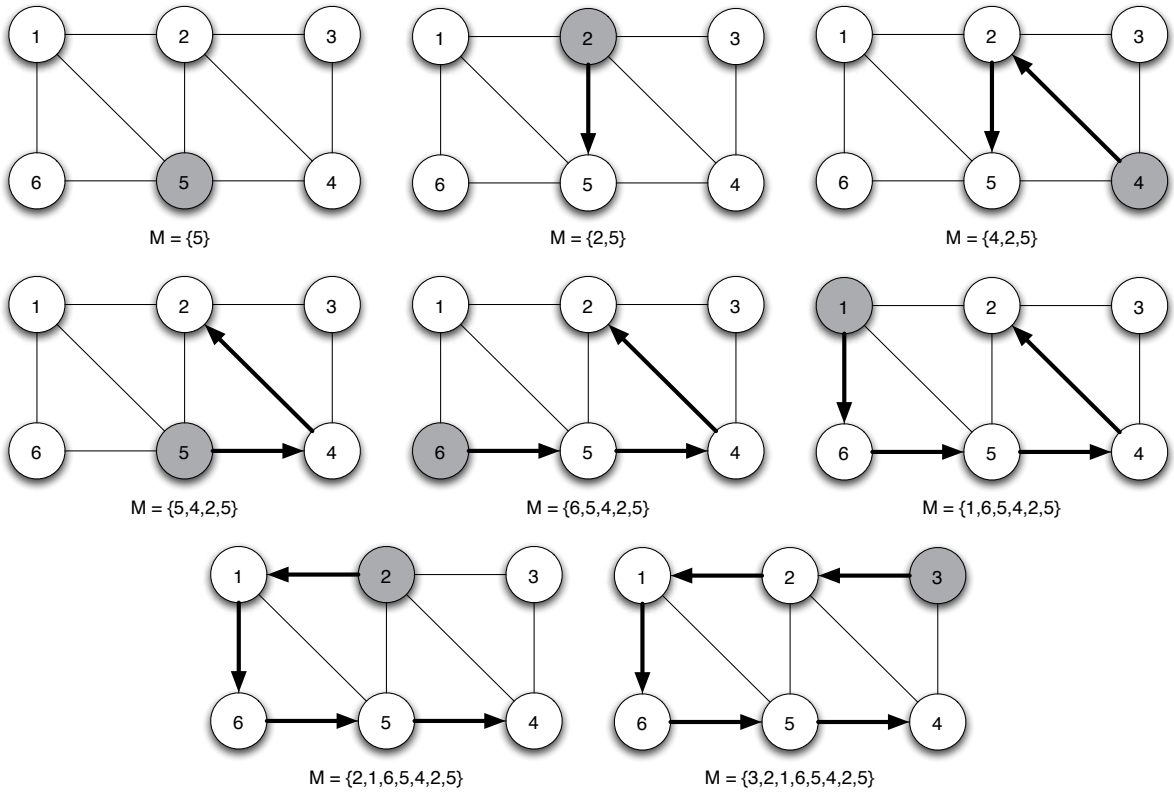


FIG. 4.1 – Arbre localement construit successivement par chacun des sites possédant le jeton

Cet algorithme a donc l'avantage de construire une structure adaptative. En contrepartie, l'image de cette structure n'est visible que par le site qui détient le jeton.

Le mot circulant permet donc à chaque site de calculer un arbre couvrant dont il est la racine à l'aide de l'Algorithme 6.

On appelle *occurence* un élément du mot circulant. Pour construire ces algorithmes, nous utilisons les procédures suivantes :

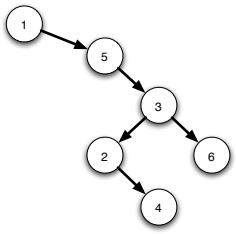
- $M[k]$: retourne le k-ième élément du mot M.
- $taille(M)$: retourne la taille du mot M.

- $Ajout\_arbre(occ, parent, arbre)$: cette fonction ajoute le nœud occ dans l'arbre $Arbre$ comme fils du nœud $parent$.
- $Premiere\_occurence(occ, M)$: retourne l'indice de la première occurence de l'élément occ dans le mot M.
- $Supprimer(M, ind1, ind2)$: supprime les éléments positionnés entre l'indice $ind1$ et $ind2$ du mot M.
- $liees(a, b, m)$: retourne $vrai$ si les occurrences a et b du mot m sont liées (cf. Définition 4.2).
- $Gauche(M, k)$ retourne la partie située à gauche de k dans le mot M, c'est-à-dire le mot $M[0], \ldots, M[k]$.
- $Droite(M, k)$ retourne la partie située à droite de k dans le mot M, c'est-à-dire le mot $M[k], \ldots, M[taille(M) - 1]$.
- $placer\_racine(A, r)$: positionne la racine de l'arbre A sur le nœud r.
- $identites(M)$: retourne l'ensemble des identités contenues dans le mot M.
- $ajoute(p, M)$: ajoute l'identité p en tête du mot M.

L'Algorithme 6 permet au site qui détient le jeton, de calculer un arbre couvrant.

Algorithme 6 Procédure : Construction_arbre_depuis_mot(M : mot) retourne A : Arbre

```
A = ∅
pour k = 0 à taille(M) − 1 faire
  si M[k] ∉ A alors
    si A = ∅ alors
      placer_racine(A, M[k])
    sinon
      Ajout_arbre(M[k], M[k − 1], A)
    fin si
  fin si
fin pour
retourner A
```



Soit $M = 1, 5, 3, 2, 3, 6, 3, 2, 4$ le mot produit par la circulation du jeton. Le site 1 peut alors calculer à partir du mot M l'arbre ci-contre.

FIG. 4.2 – Construction de l'arbre enraciné sur le site 1

A partir du moment où tous les sites ont été visités, le site détenteur du jeton est en mesure de calculer un chemin de lui-même vers tous les autres sites.

4.2.2 Réduction du mot circulant

Le jeton circule perpétuellement et la taille du mot peut donc croître à l'infini. Mais un certain nombre de techniques de réduction permettent de borner la taille du mot circulant.

On observe des redondances d'informations ou l'inutilité de certaines autres pour la construction d'un arbre couvrant. [Fla01] propose deux techniques pour réduire le contenu du mot en coupant certaines informations inutiles pour la construction de l'arbre : *la coupe terminale et la coupe interne.* Il commence par définir ce qu'est une information utile à la construction de l'arbre :

Définition 4.1 (Occurrence constructive) *Une occurrence est dite constructive si elle permet d'ajouter un noeud comme feuille dans l'arbre construit ou comme père d'une nouvelle feuille dans l'arbre (l'occurrence est dite alors* constructive père*).*

Les occurrences constructives sont celles utilisées par l'Algorithme 6 dans les procédures $placer\_racine()$ et $Ajout\_arbre()$.

Coupe terminale

Le principe de la coupe terminale est de supprimer toutes les occurrences non constructives si elles sont placées à la fin du mot circulant.

Remarque 4.1 *L'occurrence d'un noeud k à la fin du mot est une occurrence constructive, si et seulement s'il n'existe pas d'autre occurrence du noeud k dans le reste du mot.*

Lemme 4.1 *La construction de l'arbre étant basée sur les occurrences constructives, la suppression successive des occurrences non constructives placées en fin de mot, ne modifie pas la construction de l'arbre.*

Ceci permet d'éliminer les occurrences non constructives à la fin du mot. Par exemple $M = 1, 2, 5, 6, 3, 4, 5, 2, 1$ peut être réduit à $M = 1, 2, 5, 6, 3, 4$ sans modifier l'arbre construit.

Voici l'algorithme réalisant la coupe terminale.

Algorithme 7 Procédure : Coupe_terminale(M : mot)

tant que $premiere\_occurence(M[taille(M) - 1], M) \neq (taille(M) - 1)$ **faire**
  $supprimer(M, taille(M) - 1, taille(M) - 1)$
fin tant que

La taille du mot avec la coupe terminale dépend du temps de retour ($h(i, i)$) de la marche aléatoire sur le graphe. Pour la plupart des graphes, le temps de retour est borné par $O(n^3)$, la taille moyenne du mot est donc bornée par $O(n^3)$.

Coupe interne

Il existe aussi des occurrences non constructives situées à l'intérieur du mot. Par exemple le mot $M = 3, 6, 1, 2, 5, 6, 3, 4, 5, 2, 1$ peut être réduit en ne conservant que les occurrences constructives : $M = 3, 6, 1, 2, 5, 3, 4$ (nous ne conservons en fait que les nouveaux noeuds et les pères des nouveaux noeuds).

Remarque 4.2 *Soit i un élément du mot M à la position k, si i est présent avant dans le mot et si l'élément suivant i (à la position $k+1$) est aussi présent avant dans le mot (i.e. dans $Gauche(M, k)$), alors i à la position k n'est pas une occurrence constructive.*

Lemme 4.2 *Il est possible d'éliminer les occurrences internes et terminales du mot si elles ne sont pas constructives.*

Lemme 4.3 *La construction de l'arbre étant basée sur les occurrences constructives, la suppression successive des occurrences non constructives dans le mot, ne modifie pas la construction de l'arbre.*

Définition 4.2 *Occurrences liées*
Deux occurrences (T[k],T[k+1]) sont dites liées *si et seulement si $T[k]$ est père de $T[k+1]$.*

Remarque 4.3 *Deux occurrences $(T[k], T[k+1])$ ne sont pas liées si et seulement si l'élément $T[k+1]$ apparaît avant dans le mot.*

Remarque 4.4 *Cet algorithme ne respecte pas la topologie du réseau : si les occurrences des noeuds i et j sont consécutives dans le mot cela ne signifie pas forcément que i et j sont voisins dans le graphe.*

Algorithme 8 Procédure : Coupe_interne(M : mot)

```
Visite ⟵ ∅
pour k = 0 à k = taille(M) − 1 faire
  si k ≠ taille(M) − 1 alors
    si M[k] ∈ Visite et M[k + 1] ∈ Visite alors
      Supprimer(M, k, k)
    fin si
  sinon
    si M[k] ∈ Visite alors
      Supprimer(M, k, k)
    fin si
  fin si
  Visite ⟵ Visite ∪ M[k]
fin pour
```

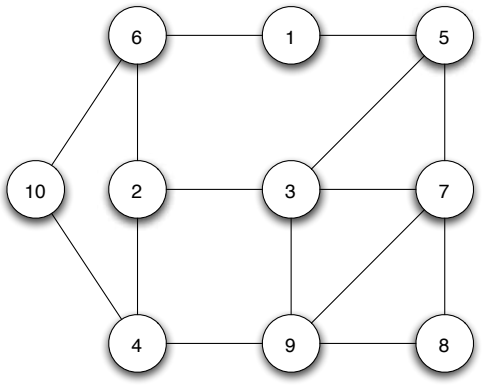
Le mot le plus long possible ne peut être composé que d'une succession d'occurrences constructives feuilles et d'occurrences constructives pères. Ainsi le plus long mot qui peut être construit a une taille de $2n - 1$, où n est le nombre de nœuds du réseau.

4.3 Structure virtuelle et tolérance aux pannes

Nous avons utilisé le mot circulant afin de centraliser l'information topologique d'un réseau au sein d'un jeton circulant aléatoirement. Cela permet au site détenteur du jeton, d'utiliser les informations du mot circulant afin d'effectuer ses actions en ayant une connaissance partielle (mais peut-être obsolète) de la topologie du réseau. Nous conservons les informations qui permettent au site de construire un arbre couvrant, et nous utilisons la coupe interne présentée en Section 4.2.2 pour réduire la taille du jeton. Nous nous intéressons particulièrement à la correction du contenu du mot circulant afin d'éviter, quand cela est possible, la transmission d'informations topologiques erronées. Nous présentons dans un deuxième temps une alternative au mot circulant. Cette alternative propose principalement une optimisation de la taille du jeton.

4.3.1 Auto-correction du contenu du mot circulant

Nous avons mis en place une procédure appelée test interne, qui vérifie **localement** la cohérence du mot circulant à partir des informations dont dispose le site qui effectue ce test. Le test parcourt le contenu du mot en vérifiant que chaque élément qui suit l'identité du site visité dans le mot apparaît bien dans le voisinage de ce site. Si une information se révèle erronée, le test coupe le mot circulant, en ne conservant que la partie qui précède cette information (partie du mot la plus récemment mise à jour).



Exemple de test interne : le jeton est situé sur le site 2 et contient le mot circulant $M = 2, 4, 10, 6, 1$. Supposons que le mot contenu dans le jeton soit "corrompu" au tour suivant. On obtient le mot $M = 3, 2, 4, \mathbf{8}, 6, 1$. Par la Remarque 4.3, les éléments 4 et 8 sont supposées être des occurences liées. Mais le site 8 n'appartient pas au voisinage du site 4. Donc à la prochaine visite du site 4, le mot sera coupé, et il restera $\cdots, 3, 2, 4$ comme contenu du mot circulant.

FIG. 4.3 – Test interne

L'algorithme suivant permet de réaliser le test interne sur le site d'identité p.

Algorithme 9 Procédure : Test_interne(M : Mot)

pour $k = 0$ à $k = taille(M) - 1$ **faire**
  si $(M[k] = p) \wedge (liees(M[k], M[k+1], M))$ **alors**
    si $M[k+1] \notin Vois_p$ **alors**
      $M \longleftarrow Gauche(M, k)$
    fin si
  fin si
fin pour

Le test interne est réalisé sur le mot à chaque fois que le jeton se déplace. Grâce à la propriété de *couverture* d'une marche aléatoire, tous les sites sont visités en temps fini, et donc toutes les *incohérences* du mot sont supprimées.

Sans l'occurrence de pannes, après C unités de temps, le jeton a visité à nouveau tous les sommets. Chaque sommet possède donc un père et donc le contenu du mot permet de calculer un arbre couvrant adaptatif en adéquation avec la topologie du réseau.

4.3.2 Réduction de la taille du jeton

Nous avons introduit dans [BBF04a], une autre représentation d'un arbre couvrant. Cette représentation est basée sur une table des relations père-fils et permet de réduire la taille du jeton de $2n - 1$ à n. L'exemple suivant expose une table de relations père-fils et son arbre associé.

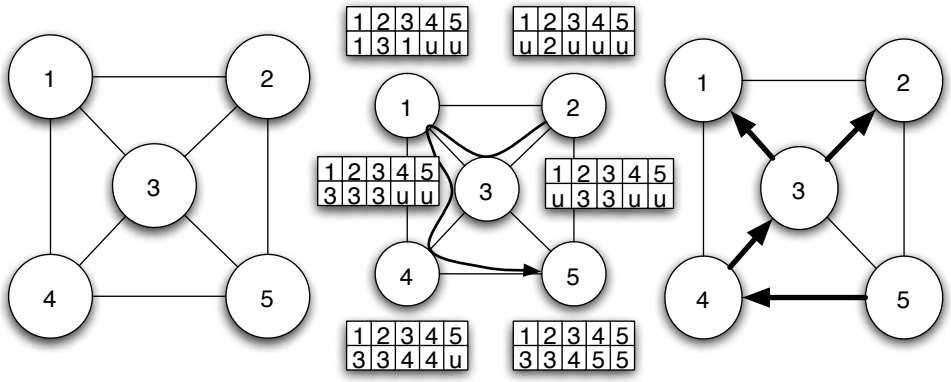


FIG. 4.4 – Réseau, Table des relations père-fils mise à jour au fur et à mesure des déplacements aléatoires du jeton, et Arbre construit sur le site 5

La construction de l'arbre est effectuée par la mise à jour de la structure de donnée stockée dans le jeton à chacun de ses déplacements. Si le jeton se déplace du site *emetteur* vers le site p, la mise à jour est effectuée par les actions suivantes :

Algorithme 10 Algorithme de mise à jour du jeton sur le site p

$jeton.table[p] \longleftarrow p$
$jeton.table[emetteur] \longleftarrow p$
envoyer $jeton$ à un site choisi uniformément au hasard parmi $Vois_p$

où $jeton.table$ est la structure de donnée représentant l'arbre stocké dans le jeton.

Les reconfigurations topologiques peuvent produire un jeton incohérent (*i.e.* une arête de l'arbre calculé n'apparait pas dans le graphe de communication). Chaque fois qu'un site reçoit le jeton, il vérifie et corrige localement la structure de donnée en appliquant la procédure de *test interne* (redéfinie ici pour la représentation par table)

Algorithme 11 Procédure : test_interne(table : tableau)

```
pour tout i = 0 à 𝒩 faire
  liste ⟵ ∅
  si (table[i] = p) ∧ (i ∉ {Vois_p ∪ {p}}) alors
    liste ⟵ liste ∪ {i}
    pour tout h ∈ liste faire
      pour j = 0 à 𝒩 faire
        si table[j] = h alors
          table[j] = indefini
          liste ⟵ liste ∪ {j}
        fin si
      fin pour
      liste ⟵ liste − {h}
    fin pour
  fin si
fin pour
```

De même que pour le mot circulant, la structure est assurée en moyenne après C unités de temps.

4.4 Mot circulant dans un réseau orienté

4.4.1 Introduction

Nous adaptons, dans cette section, la gestion du mot circulant aux réseaux orientés. Nous entendons par "réseau orienté", un réseau où les canaux de communication ne sont plus bidirectionnels. Notre objectif est de centraliser, au sein du mot circulant, une image de la topologie du réseau d'interconnexion qui permet le calcul d'une structure couvrante. Le mot circulant doit permettre au site qui le détient de calculer un arbre couvrant dont la racine est arbitrairement choisie dans le mot circulant.

Les réseaux orientés peuvent inclure aussi bien les réseaux ad-hoc, où la différence de portée entre les ondes radio de chacun des nœuds implique une telle situation, que les

réseaux filaires où peuvent être développées différentes stratégies de sécurité telles que la mise en place d'un pare feu ou la translation d'adresses IP.

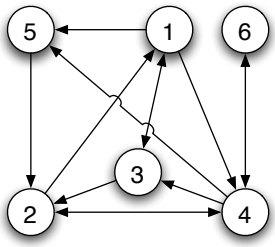
Notre algorithme aura le même schéma général que précédemment : un jeton circule aléatoirement sur le réseau et collecte de l'information topologique à chaque déplacement. Par contre toute la gestion interne du mot circulant (la stratégie de réduction, la construction de la structure couvrante, ...) devra être adaptée pour répondre aux contraintes d'un réseau orienté. En effet, nos précédents algorithmes considéraient des arêtes non orientées. Ce n'est plus le cas ici : si A est voisin de B et B n'est pas voisin de A, alors B peut être père (prédécesseur) de A mais A ne peut plus être père de B.

Nous développons la gestion du mot circulant autour de la notion de *cycle constructeur* (que nous définissons dans la Section 4.4.3).

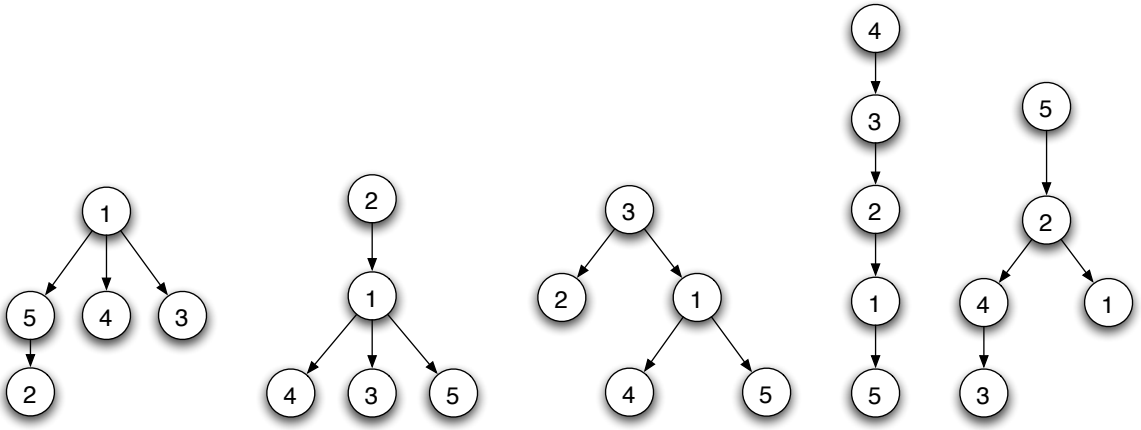
Nous commençons donc par présenter notre solution de manière informelle en illustrant son fonctionnement sur des exemples. Ensuite, nous proposons les algorithmes relatifs à la gestion du mot circulant.

4.4.2 Description informelle

Nous utilisons comme précédemment un jeton contenant un mot circulant qui parcourt le réseau d'interconnexion en collectant des informations topologiques. Ces informations doivent permettre la construction d'un arbre couvrant dont la racine est arbitrairement choisie. Néanmoins comme précédemment, le jeton circule perpétuellement, et la taille du mot doit impérativement être réduite. Au cours de la réduction, nous devons conserver les informations nécessaires à la construction de ces arbres.



Considérons le mot M construit successivement par le parcours aléatoire du jeton dans le réseau ci-contre. A ce stade de l'exécution, on a par exemple $M = 4, 1, 3, 1, 2, 3, 4, 2, 5, 1$. Voici les arbres que le site 4 peut construire :



L'algorithme fonctionne en identifiant et en ajoutant successivement les branches contenues dans le mot. Par exemple, pour le calcul de l'arbre enraciné en 1, l'algorithme (cf. Algorithme 12) place la racine de l'arbre à 1. Puis il identifie toutes les branches débutant en 1, c'est-à-dire $4 \leftarrow 1$, $3 \leftarrow 1$ et $2 \leftarrow 5 \leftarrow 1$ et les ajoute à la racine. Si des identités dans le mot n'apparaissent pas encore dans l'arborescence, l'algorithme recherche les branches débutant par les identités ajoutées au tour précédant, pour les inclure dans l'arbre.

Notre technique de réduction doit donc conserver toutes les informations topologiques nécessaires à la construction d'un arbre enraciné arbitrairement.

Nous observons qu'il existe un position particulière appelée position minimale (cf. Définition 4.6). A partir de cette position et des informations les plus récentes positionnées à gauche de cette position, nous pouvons construire un arbre couvrant enraciné sur cette position minimale. Par exemple, le mot $M = 4, 1, 3, 1, 2, 3, 4, 2, 5, 1$ a pour position minimale l'élément 5 positionné en 9ème place.

Nous remarquons que l'information pertinente correspond à la collecte d'information dans le mot, telle qu'il existe un chemin entre tout couple de site (c'est-à-dire une composante fortement connexe). Par exemple, le mot $M'' = 1, 2, 3, 4, 2, 5, 1$ nous permet de construire des arbres couvrants dont la racine peut être tout sommet contenu dans ce mot[1]. Une telle séquence d'occurrences d'identités dans le mot s'appelle *cycle constructeur* (cf. Définition 4.4). Nous remarquons que la position minimale est toujours comprise dans le cycle constructeur. Dans notre exemple, M contient deux cycles constructeurs : $1, 3, 1, 2, 3, 4, 2, 5, 1$ et $1, 2, 3, 4, 2, 5, 1$.

Le rôle de l'algorithme est donc de maintenir, au sein du mot circulant, un cycle constructeur. Néanmoins conserver le même cycle constructeur tout au long du déroulement de notre algorithme, nuirait au caractère "adaptatif" que nous recherchons. Il convient donc de le faire évoluer étape par étape pour garantir la pérennité des informations qu'il contient. Nous proposons quand cela est possible de modifier le cycle constructeur par *"décalage vers la gauche"* de la borne droite du cycle constructeur, c'est-à-dire vers les informations les plus récemment ajoutées. Ainsi, nous garantissons la mise à jour périodique de la structure maintenue. Par exemple, pour le mot $M = 4, 1, 3, 1, 2, 3, 4, 2, 5, 1$ vu précédemment, si le jeton se déplace ensuite vers le site 5, nous obtenons donc le mot circulant $M* = 5, 4, 1, 3, 1, 2, 3, 4, 2, 5, 1$. Clairement, ce mot circulant contient 3 cycles constructeurs : $CC_1 = 1, 3, 1, 2, 3, 4, 2, 5, 1$, $CC_2 = 1, 2, 3, 4, 2, 5, 1$ et $CC_3 = 5, 4, 1, 3, 1, 2, 3, 4, 2, 5$. A ce stade de l'exécution, notre méthode choisit de maintenir CC_3 puisqu'il contient les informations les plus récentes.

Les cycles constructeurs peuvent aussi contenir de l'information redondante. Par exemple $CC_1 = 1, 3, 1, 2, 3, 4, 2, 5, 1$ contient le cycle $1, 3, 1$ qui pourrait être supprimé puisque $CC_2 = 1, 2, 3, 4, 2, 5, 1$ est aussi un cycle constructeur. Le cycle $1, 3, 1$ n'apporte donc aucune information capitale dans la construction des arbres couvrants. Un tel cycle est appelé un *cycle non constructeur*(cf. Définition 4.5). Afin de réduire la taille du cycle constructeur, notre algorithme effectue une détection des cycles non constructeurs, et, le cas échéant, les supprime. Ainsi nous montrons que la taille du cycle constructeur est bornée par $n^2/4+n$ (cf. Lemme 4.4). La détection des cycles non constructeurs est effectuée

[1]On appelle abusivement une telle configuration un cycle dans ce mot

de la manière suivante : notre algorithme recherche tous les cycles inclus dans le cycle constructeur et vérifie pour chacun d'eux que les identités qu'il contient, apparaissent en dehors de ce cycle. Si c'est effectivement le cas, alors ce cycle est non constructeur. Par exemple $CC_1 = 1, 3, 1, 2, 3, 4, 2, 5, 1$ contient les cycles suivants : $C_1 = 1, 3, 1$, $C_2 = 3, 1, 2, 3$, $C_3 = 1, 2, 3, 4, 2, 5, 1$ et $C_4 = 2, 3, 4, 2$. Clairement C_3 et C_4 ne sont pas des cycles non constructeurs puisque C_3 contient 4 et 5 qui n'apparaissent qu'à l'intérieur de C_3 et C_4 contient 4 qui n'apparaît qu'à l'intérieur de C_4. Par contre C_1 et C_2 sont des cycles non constructeurs : C_1 contient l'identifiant 3 qui apparaît aussi à l'extérieur de C_1 et C_2 contient les identifiants 1 et 2 qui apparaissent à l'extérieur de C_2. Néanmoins ces deux cycles non constructeurs sont imbriqués, si on supprime l'un de ces cycles, l'autre se retrouve cassé et ne peut plus être supprimé. Si l'algorithme supprime C_1, le cycle constructeur devient alors $1, 2, 3, 4, 2, 5, 1$ et C_2 n'existe plus. Si ces deux cycles n'avaient pas été imbriqués, notre algorithme les aurait supprimés tous les deux. Dans le cas présent, notre algorithme supprime C_2 puisque C_2 contient les informations les plus anciennes, le cycle constructeur devient donc $CC = 1, 3, 4, 2, 5, 1$.

La partie située à droite du cycle constructeur peut être supprimée, nous avons montré comment réduire le cycle constructeur sans perdre d'information. Il nous reste maintenant à regarder ce qui se passe à gauche d'un cycle constructeur. Dans le mot $M * * = 4, 1, 3, 4, 2, 5, 1$ (en fait le mot M auquel nous avons appliqué la réduction de cycle non constructeur), nous avons le cycle constructeur $CC = 1, 3, 4, 2, 5, 1$ et l'identité 4 placée devant le cycle constructeur. Nous appelons cette partie la tête du mot. Elle peut croître considérablement avant que le cycle constructeur ne soit modifié. Dans notre exemple, après un certain nombre de déplacements aléatoires du jeton, le mot devient $M * ** = 3, 1, 2, 3, 4, 1, 3, 4, 2, 5, 1$. Si la tête de mot est nécessaire pour permettre le renouvellement du cycle constructeur, elle ne doit pas pour autant croître de manière trop importante. Nous proposons donc de réduire systématiquement les cycles apparaissant dans la tête de mot, certes, au risque de perdre des mises à jour d'informations topologiques. Ainsi $M * **$ se réduit à $3, 4, 1, 3, 4, 2, 5, 1$. Les opérations de réduction des cycles non constructeurs doivent en fait s'appliquer sur le cycle constructeur et sur la tête du mot de manière séparée.

A l'initialisation de l'algorithme, les réductions s'effectuent sur la tête du mot jusqu'à l'obtention du premier cycle constructeur.

Un dernier cas de figure reste à gérer. Considérons un jeton contenant un cycle constructeur (par exemple $M * * = 4, 1, 3, 4, 2, 5, 1$ avec le cycle constructeur $CC = 1, 3, 4, 2, 5, 1$). Lorsque le jeton visite un nouveau site, l'information contenu dans le mot n'est plus un cycle constructeur. Dans notre exemple, M devient $\mathbf{6}, 4, 1, 3, 4, 2, 5, 1$. Une solution simple serait de considérer le contenu de M comme une tête de mot sans cycle constructeur, et de dérouler l'algorithme comme durant l'initialisation. Nous proposons une solution plus efficace. Tant qu'un nouveau cycle constructeur incluant le nouveau site 6 n'est pas construit, l'algorithme continue de considérer $1, 3, 4, 2, 5, 1$ comme un cycle constructeur. Dés que possible, l'algorithme réécrit directement le mot sous forme d'un "vrai cycle constructeur" incluant le site 6. Par exemple si le jeton se déplace en 4 nous pouvons utiliser les informations contenues dans le mot $4, 6, 4, \mathbf{1}, \mathbf{3}, \mathbf{4}, 2, 5, 1$, pour le réécrire en $4, 6, 4, 1, 3, 4, 2, 5, \mathbf{1}, \mathbf{3}, \mathbf{4}$ c'est-à-dire en insérant le chemin de 4 vers 1.

Nous décrivons maintenant notre méthode de manière plus formelle en présentant nos

algorithmes de gestion du mot circulant.

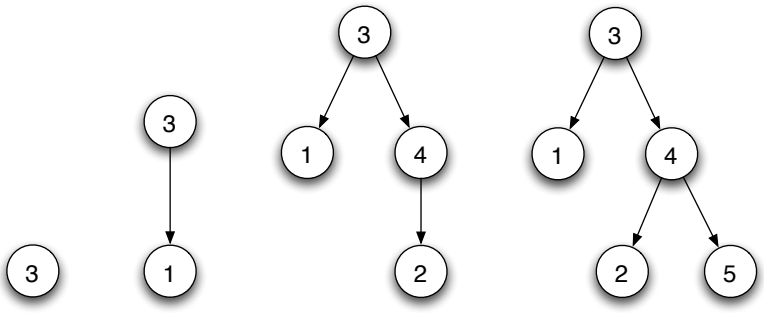
4.4.3 Description formelle de l'algorithme

Construction des structures couvrantes

L'algorithme que nous avons appliqué dans le cas des réseaux "classiques" (Algorithme 6) ne peut convenir : l'ajout d'un nœud dans l'arbre s'effectue en lisant le mot de la gauche vers la droite (*i.e.* en empruntant le chemin inverse du jeton). Une telle stratégie n'est pas envisageable dans un réseau orienté, puisque, selon le parcours du jeton, son trajet ne peut plus forcément être inversé. Notre algorithme doit donc construire l'arbre en suivant le parcours du jeton, c'est-à-dire, dans le mot circulant, de la droite vers la gauche. Comme nous l'avons expliqué dans la section précédente, l'algorithme identifie successivement les branches de l'arbre, et les ajoute dans l'arbre construit.

L'algorithme (cf. Algorithme 12) commence par positionner la racine de l'arbre à r. Puis il recherche en parcourant le mot de la gauche vers la droite, la première occurrence d'une identité contenue dans l'arbre. Une fois cette identité trouvée, l'algorithme écrit dans l'arbre le parcours qu'a effectué le jeton (cette fois ci en balayant le mot de la droite vers la gauche).

Exemple : On cherche à calculer un arbre couvrant enraciné en 3 à partir du mot $M = 1, 3, 2, 4, 3, 5, 4$. L'algorithme parcourt le mot en recherchant la première occurrence de 3 (le seul élément de l'arbre pour le moment), et, une fois celle-ci trouvée, ajoute la branche $1, 3$ à l'arbre. L'algorithme continue alors son parcours, et trouve un deuxième 3. Il ajoute donc la branche $2, 4, 3$ dans l'arbre. L'algorithme termine son parcours, et trouve un 4. Il ajoute donc la branche $5, 4$ dans l'arbre.



Algorithme 12 Procédure : Construit_arbre_couvrant(M : Mot , r : nœud) retourne A : Arbre

```
placer_racine(A, r)
i ← 0
pos ← 0
tant que i < taille(M) − 1 faire
  /* Recherche de nœud déjà ajouté dans l'arbre */
  tant que (i < taille(M) − 1) ∧ (M[i] ∉ A) faire
    i ← i + 1
  fin tant que
  si M[i] ∈ A alors
    /* Ajout d'une nouvelle branche enracinée en i */
    j ← i − 1
    tant que j ≥ pos faire
      si M[j] ∉ A alors
        Ajout_arbre(M[j], M[j + 1], A)
      fin si
      j ← j − 1
    fin tant que
  fin si
  i ← i + 1
  pos ← i
fin tant que
retourner A
```

Réduction du mot

Nous avons identifié précédemment un motif qui permet le calcul d'un arbre couvrant enraciné sur un site arbitraire. Ce motif appelé un *cycle constructeur* est en fait une composante fortement connexe regroupant les identifiants de tous les sites visités par le jeton. Notre gestion du mot circulant est donc essentiellement basée sur le cycle constructeur.

Définition 4.3 *Le mot M contient un cycle, s'il existe $(i, j) \in \{1, \ldots, taille(M)\}^2$, $i < j$ et $M[i] = M[j]$. Nous notons ce cycle $\mathcal{C}(i, j)$.*

Définition 4.4 *Un cycle $\mathcal{C}(i, j)$ dans le mot M est appelé constructeur si :*

$$\forall k \in identites(M), \exists l \in \{i, \ldots, j\} | M[l] = k$$

Un tel cycle est noté $\mathcal{C}_c(i, j)$.

Toutes les informations à droite du cycle constructeur sont des informations plus anciennes que les informations contenues dans le cycle constructeur, et ne sont pas utilisées pour la construction d'arbres. Elles sont donc totalement inutiles et peuvent être supprimées.

Comme précisé dans la section précédente, nous réduisons aussi la taille de ce cycle constructeur, en supprimant les informations redondantes à l'intérieur de ce cycle. Néanmoins il n'est pas toujours possible de supprimer ces informations facilement : en supprimant un sous-mot $M[i], \ldots, M[j]$ à l'intérieur du cycle constructeur, nous devons vérifier que les liens $(M[j+1], M[i-1])$ existe dans le mot et donc dans le réseau d'interconnexion. Une bonne alternative est de supprimer un cycle $\mathcal{C}(i, j)$, parce que le lien $M[j], M[i-1]$ existe (car $M[j] = M[i]$). Notre méthode est donc basée sur la recherche des cycles dont les informations sont inutiles. Un tel cycle est appelé un *cycle non constructeur*.

Définition 4.5 *Un cycle $\mathcal{C}(i, j)$ est appelé non-constructeur si :*

$$\forall k \in [i, j[, \exists l \in [0, i[\cup[j, taille(M)]|M[k] = M[l]$$

Algorithme 13 Procédure : Reduction_cycle_non_constructeur(M : Mot)

```
V ← ∅
pos ← taille(M) − 1
tant que pos > 1 faire
  V ← V ∪ M[pos]
  /* Recherche de cycle C(i, pos) */
  i ← pos − 1
  V_C ← ∅
  tant que (i > 0) ∧ (M[i] ≠ M[pos]) faire
    si M[i] ∉ V alors
      V_C ← V_C ∪ M[i]
    fin si
    i ← i − 1
  fin tant que
  si M[i] = M[pos] alors
    /* Cycle trouvé : recherche des éléments de V_C à gauche de i */
    j ← i − 1
    tant que (j ≥ 0) ∧ (V_C ≠ ∅) faire
      si M[j] ∈ V_C alors
        V_C ← V_C/M[j]
      fin si
      j ← j − 1
    fin tant que
    si V_C = ∅ alors
      /* Le cycle C(i, pos) est un cycle non-constructeur */
      supprimer(M, i + 1, pos)
      pos ← i
    sinon
      pos ← pos − 1
    fin si
  sinon
    pos ← pos − 1
  fin si
fin tant que
```

Notre algorithme de suppression de cycle non-constructeur parcourt le cycle constructeur de la droite vers la gauche (il supprime ainsi les informations les plus anciennes) en recherchant un cycle. L'algorithme vérifie si toutes les identités contenues dans ce cycle apparaissent aussi à l'extérieur de ce cycle. Nous sommes alors dans le cas d'un cycle non-constructeur, l'algorithme le supprime et continue son parcours. S'il existe une identité qui n'apparaît que dans ce cycle, alors ce cycle ne doit pas être supprimé. Nous obtenons alors l'Algorithme 13

Nous réussissons grâce à cette technique de réduction à borner la taille du cycle constructeur.

Lemme 4.4 *La taille du cycle constructeur est, dans le pire des cas, égale à* $\frac{n^2}{4} + n$.

Démonstration Soit m l'identité apparaissant le plus fréquemment dans le cycle constructeur, et k le nombre d'occurrences de m dans le cycle constructeur. Il y a donc $k-1$ cycles qui ne sont pas non-constructeurs dans le cycle constructeur (si ces cycles étaient non-constructeurs, ils auraient été supprimés), donc $k-1$ identités qui n'apparaissent qu'une unique fois dans le cycle constructeur et $n-(k-1)$ identités qui peuvent apparaître au plus k fois (par la définition de k). La taille maximale du mot peut donc s'écrire en fonction de k :

$$T(k) = (k-1) + (n-(k-1)) \times k = -k^2 + (n+2)k - 1$$

$T(k)$ est croissante lorsque $T'(k)$ est positive, et admet un extremum lorsque $T'(k)$ s'annule :

$$\begin{aligned} & T'(k) \geq 0 \\ \iff & -2k + n + 2 \geq 0 \\ \iff & k \leq \frac{n+2}{2} \end{aligned}$$

La taille admet un maximum en :

$$k = \frac{n+2}{2}$$

$$\begin{aligned} T_{max} &= \begin{cases} T(\frac{n+2}{2}) & \text{si } n \text{ pair} \\ \max\left\{T\left(\frac{n+1}{2}\right), T\left(\frac{n+3}{2}\right)\right\} & \text{si } n \text{ impair} \end{cases} \\ &= \begin{cases} -\left(\frac{n+2}{2}\right)^2 + (n+2) \times \frac{n+2}{2} - 1 \\ \max\left\{\left(-\left(\frac{n+1}{2}\right)^2 + (n+2)\left(\frac{n+1}{2}\right) - 1\right), \left(-\left(\frac{n+3}{2}\right)^2 + (n+2)\left(\frac{n+3}{2}\right) - 1\right)\right\} \end{cases} \\ &= \begin{cases} \frac{n^2+4n}{4} \\ \max\left\{\left(\frac{n^2+4n-1}{4}\right), \left(\frac{n^2+4n-1}{4}\right)\right\} \end{cases} \\ &= \begin{cases} \frac{n^2+4n}{4} & \text{si } n \text{ pair} \\ \frac{n^2+4n-1}{4} & \text{si } n \text{ impair} \end{cases} \end{aligned}$$

□

La gestion de la tête du mot (*i.e.* la partie à gauche du cycle constructeur) s'effectue en détectant les cycles à l'intérieur de celle-ci, et en les supprimant. Une telle technique de coupe réduit la taille de la tête du mot à $n-1$ (il ne peut pas y avoir de cycle).

Corollaire 4.1 *La taille du mot circulant est bornée par* $\frac{n(n+8)}{4}$.

Algorithme général

Nous détaillons maintenant la gestion du mot à travers la circulation du jeton. Nous explicitons les opérations que réalise le site à la réception du jeton. Ces opérations vont dépendre de l'identité ajoutée dans le mot mais aussi des informations déjà contenues. Avant l'ajout de la nouvelle identité, le mot circulant peut être dans trois états distincts (on parle aussi de phase de l'algorithme) :

Initialisation le mot circulant ne contient pas encore de cycle constructeur. Le mot circulant finit par atteindre l'état de maintenance (*i.e.* existence d'un cycle constructeur dans le mot), puisque le jeton finit par revenir sur le premier site visité.

Maintenance le mot circulant contient un cycle constructeur. Il permet donc au site qui le détient le calcul d'un arbre couvrant enraciné arbitrairement.

Collecte le mot circulant contient un cycle, mais les identités en tête de mot n'apparaissent pas dans ce cycle.

L'ajout de la nouvelle identité (celle du site actuellement visité) peut provoquer trois situations :

Site jamais visité L'identité ajoutée dans le mot n'apparaît qu'une fois dans celui-ci. Si le mot était en état de maintenance, il passe maintenant en état de collecte. Dans les deux autres cas, il ne change pas d'état.

Nouveau cycle constructeur L'identité ajoutée dans le mot circulant permet de le faire évoluer vers un nouveau cycle constructeur. Quel que soit l'état du mot, il passe en état de maintenance.

Autre cas L'identité ajoutée dans le mot circulant ne permet pas de construire un nouveau cycle constructeur, mais ne casse pas non l'éventuel cycle constructeur maintenu. L'état du mot circulant ne change pas sauf s'il on peut réecrire le mot circulant de manière à reformer le cycle constructeur. Dans ce cas, le mot circulant revient en phase de maintenance.

Avant de voir le détail des opérations réalisées, nous proposons un exemple récapitulatif des différent cas rencontrés :

Tableau exemple (issu du réseau présenté Section 4.4.2) Nous présentons, pour chaque état possible, un mot et lui ajoutons une identité (parmi les 3 situations que nous venons d'expliciter). Nous indiquons alors le mot produit par l'ajout d'identité (en gras) et le nouvel état du mot.

Etat du mot	**Initialisation**	**Maintenance**	**Collecte**
Mot exemple	$4,1,2,5$	$4,1,2,3,4$	$1,2,5,4,6,4$
Site jamais visité	$\mathbf{3},4,1,2,5$ Init.	$\mathbf{5},4,1,2,3,4$ Coll.	$\mathbf{3},1,2,5,4,6,4$ Coll.
Nouveau CC	$\mathbf{5},4,1,2,5$ Maint.	$\mathbf{3},4,1,2,3$ Maint.	$\mathbf{4},1,2,5,4,6,4$ Maint.
Autre cas	$\mathbf{2},4,1,2,5$ Init.	$\mathbf{2},4,1,2,3,4$ Maint.	$\mathbf{5},1,2,5,4,6,4$ Coll.

Nous en déduisons le tableau d'opérations suivant (Note : CC = Cycle Constructeur, CNC = Cycle non constructeur, CM = Cycle maintenu dans la phase de collecte) :

Etat du mot	**Situation**	**Opérations à réaliser**
Initialisation	Site jamais visité	rien à faire
	Nouveau CC	suppression des CNC dans le nouveau CC suppression de la partie à droite du nouveau CC
	Autre cas	suppression des CNC
Maintenance	Site jamais visité	rien à faire
	Nouveau CC	suppression des CNC dans le nouveau CC suppression de la partie à droite du nouveau CC
	Autre cas	suppression des cycles dans la tête du mot
Collecte	Site jamais visité	rien à faire
	Nouveau CC	suppression des CNC dans le nouveau CC suppression de la partie à droite du nouveau CC
	Autre cas	essai de réécriture d'un CC ? Succès : suppression des CNC dans le nouveau CC Echec : réduction de CNC à droite du CM

Pour expliciter les algorithmes relatifs à l'état du mot, nous introduisons la notion de *position minimale* :

Définition 4.6 *Une position est appelée position* minimale *et notée* pos_{min} *dans le mot* M*, si elle satisfait le critère suivant :*

$$pos_{min} = \min\{i \in [0, taille(M) - 1] | \forall k \in identites(M), \exists j \leq i, M[j] = k\}$$

Remarque 4.5 *La position* pos_{min} *est unique.*

L'Algorithme 14 permet le calcul de pos_{min}.

Algorithme 14 Procédure : Calcul_pos_min(M : Mot) retourne i : entier

$pos_{min} \leftarrow 0$
$visite \leftarrow \emptyset$
$i \leftarrow 1$
tant que $i \leq taille(M)$ **faire**
  si $M[i] \notin visite$ **alors**
    $visite \leftarrow visite \cup M[i]$
    $pos_{min} \leftarrow i$
  fin si
  $i \leftarrow i + 1$
fin tant que
retourner pos_{min}

Le mot en état d'initialisation ne possède pas de cycle basé sur la borne droite du mot (*i.e.* la position minimale est aussi la borne droite du mot). La seule opération à réaliser est donc la suppression des cycles non constructeurs. La formation de ceux-ci intervient

lorsque le jeton visite une identité qu'il a déjà visité auparavant (par exemple le mot $2, 1, 2, 1, 3$).

Cela nous donne, pour la phase d'initialisation, l'algorithme suivant :

Algorithme 15 Procédure : Phase_initialisation()

si $M[0] \in identites(Droite(M, 1))$ **alors**
  $Reduction\_cycle\_non\_constructeur(M)$
fin si

D'après la propriété de *percussion* d'une marche aléatoire, le jeton revient sur le premier site visité et un premier cycle constructeur est donc créé. L'algorithme continue ensuite en phase de maintenance.

La phase de maintenance a pour objectif de maintenir et de faire évoluer le cycle constructeur afin que celui-ci s'accorde avec les reconfigurations topologiques qui apparaissent dans le réseau. Suivant l'identité du site visité l'algorithme adopte un comportement différent. Le mot s'écrit de la manière suivante :

$$M = \underbrace{M[0], \ldots, M[i]}_{tete}, \ldots, M[pos_{min}], \ldots, M[taille(M) - 1]$$

et $\mathcal{C}_C(i, taille(M) - 1)$ le cycle constructeur du mot. Les identités des sites visités s'accumulent dans la tête du mot. Afin de réduire la taille de celle-ci nous effectuons une réduction des cycles non-constructeurs mais uniquement sur la tête du mot.

Le cycle constructeur ne va évoluer que si l'identité du site visité apparaît entre les positions pos_{min} et $taille(M)-1$: d'après la définition du cycle constructeur et de la position minimale, la borne droite du cycle constructeur ne peut être comprise qu'à droite de $M[pos_{min}]$ (une fois que toutes les identités des sites visités sont repérées au moins une fois sur le mot). Si un nouveau cycle apparaît, l'algorithme supprime la partie située à droite de celui-ci. Le mot circulant étant constitué uniquement du nouveau cycle constructeur, il est possible d'effectuer une recherche et une suppression des cycles non-constructeurs.

Algorithme 16 Procédure : Phase_maintenance()

si $M[0] \in identites(Droite(M, 1))$ **alors**
  si $M[0] \in [M[1], \ldots, M[i]]$ **alors**
    /* *Réduction de la tête du mot* */
    $supprimer(M, 1, j)$ avec $M[0] = M[j]$ et $j \leq i$
  sinon
    si $M[0] \in [M[pos_{min}], \ldots, M[taille(M) - 1]]$ **alors**
      /* *Nouveau cycle constructeur : réduction à droite du nouveau cycle constructeur* */
      $supprimer(M, j, taille(M) - 1)$ avec $M[j] = M[1]$ et $pos_{min} < j < taille(M) - 1$
      $Reduction\_cycle\_non\_constructeur(M)$
    fin si
  fin si
fin si

La phase de collecte a pour but de maintenir un cycle qui n'est pas constructeur afin d'aboutir au plus vite à la reconstruction d'un cycle constructeur. Le mot circulant maintenu a le format suivant :

$$M = \underbrace{M[0], \ldots, M[i]}_{tête}, \ldots, M[taille(M) - 1]$$

et $\mathcal{C}(i, taille(M) - 1)$ le cycle maintenu.

Si l'identité du site visité n'appartient pas à $\mathcal{C}(i, taille(M) - 1)$, il n'est pas encore possible de réécrire le mot de manière à obtenir un cycle constructeur. Néanmoins, l'algorithme tente de réduire les informations inutiles dans la partie située à droite du cycle.

Si maintenant l'identité du site visité apparaît dans $\mathcal{C}(i, taille(M) - 1)$, le mot a le format suivant :

$$M = M[0], \ldots, M[i], \ldots, M[j], \ldots, M[taille(M) - 1]$$

avec $M[0] = M[j]$

Il est alors possible de le réécrire en :

$$M = M[0], \ldots, M[i], \ldots, M[j], \ldots, M[taille(M) - 1], \ldots, M[j]$$

car $M[i] = M[taille(M)-1]$, l'algorithme général retourne alors en phase de maintenance.

Remarque 4.6 *La portion devant le cycle $\mathcal{C}(i, taille(M) - 1)$ contient des identités qui n'apparaissent donc pas dans $\mathcal{C}(i, taille(M) - 1)$.*

L'algorithme suivant réalise la phase de collecte

Algorithme 17 Procédure : Phase_collecte()

si $M[0] \notin [M[i], \ldots, M[taille(M) - 1]]$ **alors**
  $Reduction\_cycle\_non\_constructeur(sous\_mot(M[0], M[i-1]))$
sinon
  $\exists j \in \{i, \ldots, taille(M) - 1\} | M[j] = M[0]$
  Réécrire le mot $M = M[0], \ldots, M[i], \ldots, M[pos_{min}], \ldots, M[taille(M) - 1]$ en $M = M[0], \ldots, M[i], \ldots, M[pos_{min}], \ldots, M[taille(M) - 1], \ldots, M[j]$ avec $M[j] = M[0]$
  $Reduction\_cycle\_non\_constructeur(M)$
fin si

L'algorithme général va osciller un certain temps entre la phase de collecte et la phase de maintenance. Mais à une certaine étape de l'exécution, et si aucune reconfiguration topologique n'apparaît, l'algorithme retourne définitivement en phase de maintenance car tous les sites du réseau sont finalement visités.

Il nous reste à préciser comment l'algorithme détecte l'état du mot. Nous avons montré, pour chacune des phases, le format du mot M. Nous pouvons dire que :

1. Si pos_{min} est aussi la borne droite du mot M, alors le mot est dans l'état d'initialisation. Sinon, il existe un cycle $\mathcal{C}(i, taille(M) - 1)$ dont la borne droite est aussi la borne droite du mot.

2. Si une des identités dans $M[1], \ldots, M[i-1]$ apparaît aussi dans $\mathcal{C}(i, taille(M)-1)$, alors le mot est dans l'état maintenance (cf. Remarque 4.6). Sinon, le mot est dans l'état collecte.

Cela nous donne l'algorithme suivant :

Algorithme 18 Algorithme principal de circulation sur le site p

Reception du jeton J contenant le mot M
$ajoute(p, M)$
$pos_{min} \longleftarrow Calcul\_pos\_min(M)$
si $pos_{min} \neq taille(M) - 1$ **alors**
  Recherche de $M[i]$ à partir de $M[1]$ tel que $M[i] = M[taille(M) - 1]$
  Recherche de $M[j]$ à partir de $M[i]$ tel que $M[j] = M[1]$
  si $M[j]$ n'existe pas **alors**
    $Phase\_collecte()$
  sinon
    $Phase\_maintenance()$
  fin si
sinon
  $Phase\_initialisation()$
fin si
Envoyer J contenant M à i choisi uniformément au hasard dans $Vois_p$

Test local

Nous avons mis en place comme pour le cas des communications bidirectionnelles un test interne qui vérifie localement la cohérence du mot circulant en adéquation avec les informations localement connues par le site. L'algorithme doit donc, à cause de l'orientation des arêtes, parcourir le mot de droite à gauche en recherchant d'abord la position (pos) de l'identité du site que le jeton visite actuellement. Celà nous donne le schéma suivant :

$$M[0], \ldots, M[pos-1], M[pos], \ldots, M[taille(M)-1]$$

avec $M[0] = M[pos]$.

L'algorithme vérifie alors si $M[pos-1]$ appartient bien à $Vois_{M[pos]}$. Si c'est le cas, l'algorithme continue sa recherche avec une autre position pos telle que $M[pos] = M[0]$. Sinon l'algorithme cherche à gauche de pos la première occurrence d'une identité qui appartient à son voisinage :

$$M[0], \ldots, M[i], \ldots, M[pos-1], M[pos], \ldots, M[taille(M)-1]$$

avec $M[0] = M[pos]$, $M[pos-1] \notin Vois_{M[pos]}$ et $M[i] \in Vois_{M[pos]}$.

L'algorithme supprime alors la partie comprise entre $M[i]$ et $M[pos]$. Si ce i n'existe pas alors l'algorithme supprime toute la partie du mot à gauche de $M[pos]$.

Ces remarques permettent la construction de l'algorithme :

Algorithme 19 Procédure : Test_interne(M : Mot)

```
pos ← taille(M) − 1
tant que pos > 0 faire
  si M[pos] = i alors
    gauche ← pos − 1
    tant que (gauche > 0) ∧ (M[gauche] ≠ i) ∧ (M[gauche] ∉ Vois_i) faire
      gauche ← gauche − 1
    fin tant que
    si (M[gauche] = i) ∨ (M[gauche] ∈ Vois_i) alors
      si gauche ≠ pos − 1 alors
        supprimer(M, gauche + 1, pos − 1)
      fin si
    sinon
      si gauche = 0 alors
        supprimer(M, 0, pos − 1)
      fin si
    fin si
    pos ← gauche
  sinon
    pos ← pos − 1
  fin si
fin tant que
```

Le test interne doit être réalisé juste avant l'envoi du jeton à un site voisin.

4.4.4 Circuit hamiltonien

Nous avons montré que nous pouvions dans un réseau orienté, utiliser un mot circulant, et obtenir, grâce aux propriétés des marches aléatoires, un cycle constructeur. Ce cycle permet le calcul de chemins entre tout couple de sites. Nous allons montrer maintenant en modifiant un peu notre algorithme que nous pouvons calculer de manière probabiliste un cycle hamiltonien dans un réseau (graphe) orienté. Un circuit hamiltonien est défini par ([GJ79]) :

Définition 4.7 *Un circuit hamiltonien sur le graphe G est un circuit simple qui inclut tous les sommets de G.*

L'algorithme présenté précédemment maintient un cycle constructeur qui évolue en s'adaptant aux reconfigurations topologiques du réseau. Nous sommes dans le cadre de ce problème restreint à un graphe statique. Nous allons donc restreindre cette faculté d'adaptation dans l'optique de réduire au minimum la taille du cycle constructeur et donc d'arriver, quand cela est possible, à l'obtention d'un cycle hamiltonien.

Considérons que nous avons un cycle constructeur comprenant toutes les identités du réseau (graphe) d'interconnexion.

Le mot, à une étape où il doit évoluer, peut donc s'écrire sous la forme :

$$M = \underbrace{M[0], \ldots, M[i]}_{A}, \ldots, M[pos_{min}], \ldots, \underbrace{M[j], \ldots, M[taille(M)-1]}_{B}$$

avec $M[i] = M[taille(M)-1]$ et $M[0] = M[j]$.

A cette étape notre algorithme fait évoluer le cycle constructeur $\mathcal{C}_C(i, taille(M)-1)$ vers $\mathcal{C}_C(0, j)$, nous proposons dans le cadre de la recherche de cycle hamiltonien de n'effectuer cette évolution que si la taille de A est inférieure ou égale à celle de B. Si ce n'est pas le cas, nous allons réecrire un autre cycle constructeur en remplaçant A par B, ce qui nous donne :

$$M = M[j], \ldots, M[taille(M)-1], \ldots, M[pos_{min}], \ldots, M[j]$$

La taille du cycle constructeur va donc, étape par étape, soit stagner soit décroître, jusqu'à atteindre quand cela est possible un circuit hamiltonien.

L'adaptation de notre algorithme est donc un algorithme de type Las Vegas : quand le circuit hamiltonien existe, l'algorithme finit par le trouver, mais nous ne pouvons pas fournir de borne au temps de calcul de celui-ci : son calcul dépend directement du trajet qu'emprunte le jeton.

4.5 Conclusion

Nous avons proposé dans ce chapitre une méthode qui utilise l'adaptabilité des marches aléatoires, pour centraliser dans un jeton une image partielle de la topologie du réseau. Cette centralisation d'information est réalisée grâce au mot circulant. L'image de la topologie du réseau peut être utilisée à l'initiative du site qui possède le jeton, pour construire une structure couvrante sur le réseau. Grâce au test interne que nous avons mis en place sur chacun des sites, cette image est adaptative. L'occurrence d'une reconfiguration topologique entraîne une vision erronée de la topologie pendant un certain temps, mais celle-ci finit par être corrigée. En outre, nous avons adapté cette méthode aux réseaux orientés. Nous utilisons cette structure dans le chapitre 5, pour concevoir un algorithme auto-stabilisant de circulation de jeton. Nous fournissons ici un récapitulatif sur la taille du mot circulant.

	réseau non orienté			réseau orienté
Taille	représentation :	mot	table	$n^2/4 + 2n$
		$2n-1$	n	

Chapitre 5

Un algorithme auto-stabilisant de circulation aléatoire de jeton pour les réseaux dynamiques

Résumé : *Nous adaptons la circulation de jeton que nous avons exposée précédemment pour la rendre tolérante aux environnements fautifs. Nous présentons les solutions que nous avons apportées pour résoudre les différents problèmes induits par les pannes de communication. En particulier, nous avons introduit un mécanisme appelé* vague de réinitialisation *pour garantir le caractère auto-stabilisant de notre algorithme. Ces travaux ont fait l'objet d'une publication dans la conférence internationnale ISPA [BBF04a].*

5.1 Introduction

Dans ce chapitre, nous prenons en considération la présence de pannes transitoires dans le système.

Notre solution au problème de circulation de jeton dans un environnement dynamique repose d'une part sur l'utilisation des marches aléatoires et d'autre part sur le concept d'auto-stabilisation (cf. Section 2.3.2).

Le problème de circulation de jeton consiste à garantir d'une part, l'unicité du jeton et d'autre part, la visite de tous les sites du réseau en un temps fini. Nous sommes amenés à résoudre deux configurations particulières :

1. La situation d'absence de jeton (qui correspond à une situation d'interblocage de communication).
2. La situation où plusieurs jetons sont présents dans le système.

Dans [Var94], l'auteur propose une adaptation de l'algorithme de Dijkstra (cf. Algorithme 1) dans le modèle à passage de messages. En particulier, une circulation de jeton auto-stabilisante sur un anneau est présentée. L'interblocage de communication est résolu en utilisant une horloge de garde[1] sur un site distingué. Néanmoins, les duplications de

[1] timeout en anglais

jetons peuvent encore survenir. L'auteur introduit alors le paradigme de "counter flushing"[2] pour résoudre le problème d'occurrences multiples de jeton et ainsi obtenir un algorithme auto-stabilisant de circulation de jeton. L'idée du "counter flushing" a été reprise et adaptée par de nombreux articles de la littérature tels que [CW02, HV01].

Notre objectif est de proposer des solutions pour des réseaux dynamiques. Cela implique que la topologie du réseau est arbitraire. Dans le dernier cas (jetons multiples), les jetons se rencontrent en un temps fini (cf. Section 2.5.2) et nous décidons alors de les fusionner en un seul, comme le font Israeli et Jalfon dans [IJ90]. Dans le modèle à passage de messages, l'interblocage de communication[3] (premier cas) est un véritable problème : comment déterminer l'absence de messages. Une solution proposée dans [GM91] est d'utiliser des horloges de garde pour générer de nouveaux messages lorsque l'on "suspecte" la perte de messages.

Dans [DSW02], les auteurs utilisent un agent mobile se déplaçant comme une marche aléatoire pour effectuer de la communication de groupe auto-stabilisante dans les réseaux ad-hoc. L'agent (qui se comporte comme un jeton) est utilisé pour effectuer de la diffusion d'information. Cet article prend pour hypothèse l'existence d'un agent unique dans le système. Un mécanisme d'horloge de garde est utilisé pour la création d'un nouvel agent quand aucun agent n'existe (cas similaire à l'interblocage de communication). Les auteurs suggèrent de choisir *"une période tpi suffisamment longue (qui serait fonction du temps de couverture du graphe) pour produire un nouvel agent" et "pour éviter la création simultanée de nouveaux agents, de choisir tpi aussi en fonction de l'identifiant du processeur pi, c'est-à-dire finalement* $tpi = i \times C$ *où* i *est l'identifiant de pi et* C *le temps de couverture"*. La garantie qu'en un temps fini, il n'existe plus qu'un seul et unique agent, n'est pas clairement démontrée dans [DSW02] puisque C est défini comme le temps moyen de visite de tous les sommets.

L'idée d'avoir un site distingué est idéale pour contrôler la production de jeton s'il est possible de garantir "l'espérance de vie" de ce site. Ce n'est malheureusement pas possible dans le cas des réseaux dynamiques. Il faut utiliser une solution décentralisée. Compter les jetons est une tâche ardue : ceux-ci se déplacent *aléatoirement* dans une topologie *arbitraire*. Une autre idée serait de produire de nouveaux jetons uniquement si nécessaire à l'aide d'horloge de garde. Cette solution est réalisable si on connaît une borne sur le temps de visite par le jeton de tous les sites. C (définit Section 2.5.3) peut être un bon choix si on est sûr que chacun des sites a été visité. Ce n'est pas le cas. C est un temps moyen, et la création de nouveaux jetons en utilisant des horloges de garde initialisées à C unités de temps, n'assure pas que la production de nouveaux jetons s'effectue en l'absence de jetons dans le réseau.

Notre solution utilise une circulation aléatoire de jeton comme solution aux reconfigurations topologiques d'un réseau dynamique. Pour couvrir le problème d'interblocage de communication, la version auto-stabilisante de notre algorithme utilise une procédure totalement décentralisée d'horloges de garde. Dans une configuration où il n'existe pas de jeton, chaque processeur a indistinctement la possibilité de produire un nouveau jeton. Des créations multiples et/ou simultanées peuvent donc parfois survenir. Néanmoins, notre

[2] compter et vider en francais

[3] communication deadlock en anglais

algorithme garantit qu'en un temps fini seul un jeton finit par circuler. Pour éviter la production intempestive de nouveaux jetons, nous avons introduit un nouveau mécanisme : la vague de réinitialisation[4]. Cette vague est propagée sur un arbre adaptatif maintenu dans le jeton grâce au mot circulant. Chaque processeur maintient une horloge, et sur un déclenchement de cette horloge, produit un jeton. La vague de réinitialisation prévient alors de toute création inutile de jeton : chaque processeur déjà visité par le jeton (en fait contenu dans le mot circulant) ne déclenche pas son horloge et celle-ci est réinitialisée. Après la diffusion de la vague de réinitialisation, le seul cas où des processeurs peuvent créer de nouveaux jetons correspond à une configuration illégale (le processeur n'a pas encore été visité par le jeton). Le mécanisme de vague de réinitialisation est décrit dans la section suivante.

Hypothèses : Nous travaillons dans le modèle à passage de messages. Nous considérons des délais de communication bornés (un message est reçu au plus après Δ unités de temps ou alors est perdu) et un temps de calcul local sur un site borné (Θ unités de temps). Enfin notre algorithme requiert une borne $\mathcal{N}$ sur le nombre de sites sur le réseau.

5.2 Algorithme auto-stabilisant de circulation de jeton

Concevoir un algorithme auto-stabilisant de circulation de jeton, nécessite d'une part de gérer des configurations arbitraires (perte de jeton, jetons multiples) et d'autre part de gérer au mieux le dynamisme du réseau d'interconnexion. Nous détaillons donc les solutions que nous apportons à chacun des problèmes.

5.2.1 Circulation de jeton dans un réseau dynamique

Une circulation de jeton utilisant un déplacement similaire à une marche aléatoire, est suffisante pour résoudre le problème de circulation de jeton dans un réseau sujet à de fréquentes reconfigurations topologiques. Si le jeton est sur le site i au temps t, au temps $t+1$, il sera sur l'un des voisins j de i avec une probabilité égale à $1/\deg(i)$. Ainsi, les reconfigurations topologiques sont naturellement gérées : si un canal (i,j) devient indisponible (le réseau reste néanmoins connexe), la probabilité que le jeton traverse ce canal devient nulle et le jeton finit par atteindre le site j après le temps moyen $h(i,j)$ (défini comme le temps moyen de percussion entre i et j). L'ajout d'un canal de communication est géré de manière similaire : le voisinage est modifié et le jeton peut de nouveau emprunter ce nouveau canal pour se déplacer.

[4]reloading wave en anglais

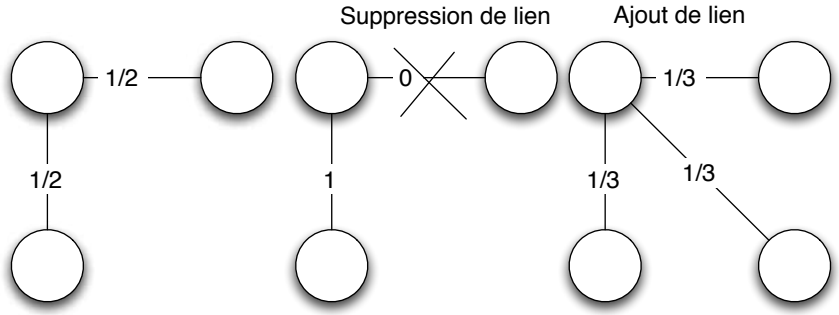


FIG. 5.1 – Adaptativité d'une marche aléatoire

5.2.2 Perte de jeton

Le jeton visite tous les sites du réseau (cf. Propriété de couverture Section 2.5.2). Mais un site donné reçoit le jeton en un temps fini mais non borné.

Par conséquent, une solution au problème d'interblocage de communication dans les réseaux dynamiques, directement inspirée de [Var94], n'est pas adaptée. Le choix d'une période suffisamment longue aux horloges de garde pour produire de nouveaux jetons, compromet la propriété de clôture de notre algorithme : malgré la présence d'un jeton il est possible qu'un autre jeton soit créé. Pour les mêmes raisons, le choix d'un site distingué qui prendrait en charge les éventuelles créations de jeton grâce à une horloge de garde, est inadapté (ce site peut parfaitement tomber en panne). Ainsi, notre algorithme est complètement décentralisé et chacun des sites maintient une horloge de garde qui permet la création périodique de nouveaux jetons. Nous initialisons les horloges de garde des différents sites à $Tmax = \max_{\forall(i,j)\in V^2}(h(i,j)) \times (\Delta + \Theta)$ unités de temps. Cette valeur correspond au pire des temps moyens pour qu'un jeton partant du site i atteigne le site j.

Néanmoins, la création de jeton reste possible, même si un jeton est déjà présent dans le réseau, puisque $h(i,j)$ est défini comme un temps moyen. Afin de résoudre ce problème, nous introduisons un nouvel outil : la *vague de réinitialisation*. Pendant l'exécution de notre algorithme (en période stable), lorsque qu'un site i est sur le point de produire un nouveau jeton, il reçoit un ordre provenant de la diffusion de cette vague. i est informé de l'existence d'un jeton dans le réseau et reçoit l'ordre de réinitialiser son horloge.

La vague de réinitialisation est propagée si les conditions suivantes sont verifiées : le jeton maintient un compteur permettant de connaître le nombre de sauts qu'il a déjà effectués depuis sa création. Sa valeur est comparée à la valeur initiale des horloges de garde ($\max_{\forall(i,j)\in V^2}(h(i,j)) \times (\Delta + \Theta)$) auquel on soustrait le temps nécessaire à la propagation de la vague (dans le pire des cas $\mathcal{N} \times (\Delta + \Theta)$: temps d'une propagation d'information sur une chaîne). Quand la valeur de ce compteur est supérieure ou égale à la dernière, le site possédant le jeton déclenche la propagation de la vague et le compteur de saut du jeton est réinitialisé à 0. Quand un site reçoit la vague, il réinitialise son horloge à $Tmax$ et continue la propagation de cette vague.

La vague de réinitialisation est définie par rapport à un jeton. Nous notons $VR(l)$ la

vague de réinitialisation associée au jeton l. Quand un site i donné reçoit un message de $VR(l)$, il notifie que le jeton l circule toujours dans le système. Ainsi $VR(l)$ empêche un site de produire une nouvelle copie du jeton l.

Nous propageons cette vague sur un arbre adaptatif. Puisque le déclenchement de la vague se fait à l'initiative du jeton, cet arbre sera maintenu sous forme de mot circulant (cf. Chapitre 4) à l'intérieur du jeton.

Dans le cas où l'arbre est un arbre couvrant sur le réseau, nous avons le schéma de fonctionnement suivant :

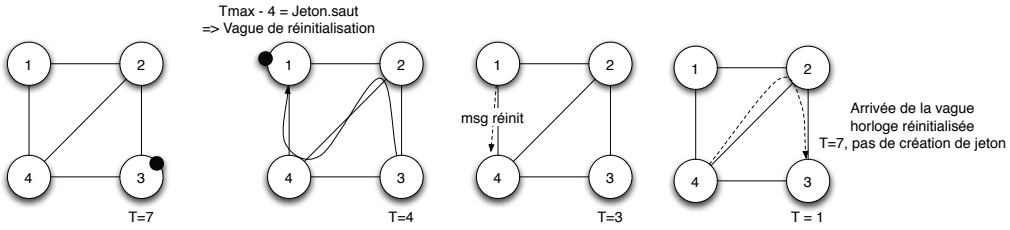


FIG. 5.2 – Exemple de propagation de la vague sur un arbre couvrant

1. Le jeton est sur le site 3 dont l'horloge est initialisée à $Tmax = 7$. Le compteur de saut du jeton est initialisé à 0.
2. Le jeton se déplace aléatoirement et, après trois sauts, arrive sur le site 1. La condition de déclenchement de la vague est vérifiée ($Tmax - 4 = jeton.saut$. 4 étant le nombre de sites du réseau). La vague est donc propagée sur l'arbre construit grâce au mot circulant (ici la chaîne $1, 4, 2, 3$).
3. Le site 4 reçoit la vague, réinitalise son horloge et continue la propagation de la vague à travers le sous-arbre $4, 2, 3$
4. La vague arrive sur le site 3 (après être passée sur le site 2), la valeur de l'horloge est réinitialisée de 1 à 7 et donc, il n'y a pas de création de jeton.

Si le mot circulant construit au sein du jeton ne permet pas la construction d'un arbre couvrant sur le réseau alors il y aura peut être un risque de création de jeton :

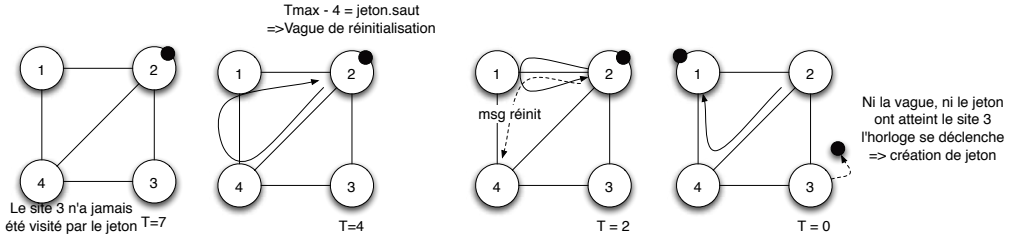


FIG. 5.3 – Exemple de propagation de la vague à partir d'un mot ne permettant pas la construction d'un arbre couvrant

1. Le site 3 n'a pas encore été visité par le jeton, les valeurs de chacune des horloges sont égales à 7.

2. Le jeton se déplace aléatoirement et, après trois sauts (4,1 et 2), revient sur le site 2. La condition de déclenchement de la vague est vérifiée ($Tmax - 4 = jeton.saut$ 4 étant le nombre de sites du réseau). La vague est donc propagée sur le sous-arbre construit grâce au mot circulant (ici la chaîne $2, 1, 4$).

3. Le site 4 reçoit la vague, réinitalise son horloge, la propagation de la vague est terminée.

4. L'horloge du site 3 se déclenche, un nouveau jeton est créé. L'horloge du site 3 est réinitialisée à $Tmax = 7$.

La vague de réinitialisation empêche donc la création de nouveaux jetons, dès lors que l'arbre construit à partir du mot circulant, est un arbre couvrant valide. Une fois tous les sommets du réseau visités, le mot circulant permet la construction d'un arbre couvrant. Ceci arrive en moyenne après $C \times (\Delta + \Theta)$ unités de temps.

5.2.3 Résoudre les situations de jetons multiples

D'autres situations peuvent survenir, en particulier, des configurations initiales où plusieurs jetons sont présents dans le réseau. De tels problèmes doivent être résolus. Dans [IJ90], les auteurs proposent une solution dans le modèle à états : un site qui a des voisins possédant un jeton, les fusionne en un seul. Dans notre modèle, le modèle à passage de messages, nous devons adapter cette solution : un site ne peut pas lire l'état de ses voisins. Notre algorithme fusionne alors tous les jetons présents sur le même site au même moment. Dans [TW91], les auteurs ont montré que quel que soit la configuration initiale, le système atteint une configuration où un seul jeton est présent, en temps polynomial ($O(n^3)$). Afin d'accélérer la convergence vers une configuration légale, notre algorithme fusionne aussi les mots circulants contenus à l'intérieur de chacun des jetons. L'algorithme construit un arbre avec les mots contenus dans les jetons fusionnés. Comme tous ces jetons sont sur le même site, ils sont enracinés sur le même sommet. Il est alors facile de construire l'union de tous ces arbres. Nous présentons ici l'algorithme qui fusionne deux jetons. Dans le cas où il y a plus de deux jetons, notre algorithme fusionne ceux-ci deux-à-deux.

L'algorithme fonctionne de la manière suivante, il construit l'arbre associé à l'un des deux mots puis parcourt l'autre mot en ajoutant les identités qui n'appartiennent pas encore à l'arbre.

Algorithme 20 Procédure : Fusion_jetons(J1 : Jeton, J2 : Jeton) retourne J : Jeton

$saut \longleftarrow max(J1.saut, J2.saut)$
$mot \longleftarrow T1.mot$
$mot' \longleftarrow T2.mot$
$Arbre \longleftarrow Construction\_arbre\_depuis\_mot(mot)$
pour $k = 0$ à $taille(mot') - 1$ **faire**
  si $mot'[k] \notin Arbre$ **alors**
    $Ajout\_arbre(mot'[k], mot'[k-1], Arbre)$
  fin si
fin pour
$J.saut \longleftarrow saut$
$J.mot \longleftarrow Construction\_mot\_depuis\_arbre(Arbre)$
retourner J

Cet algorithme utilise la procédure *Construction_arbre_depuis_mot*() qui reconstruit un mot à partir d'un arbre en respectant la coupe interne (cf. Section 4.2) :

Algorithme 21 Procédure : Construction_mot_depuis_arbre (A : Arbre) retourne m : Mot

tmp est une variable utilisée pour conserver l'identité du père des nouvelles occurences constructives
$tmp \longleftarrow \emptyset$
$m \longleftarrow mot\_vide$
Parcourir l'arbre A avec un parcours en profondeur d'abord
pour Chaque nœud visité **faire**
  si $id\_nœud\_courrant \notin m$ **alors**
    $ajoute(tmp, m)$
    $ajoute(id\_nœud\_courant, m)$
    $tmp \longleftarrow \emptyset$
  sinon
    $tmp \longleftarrow id\_nœud\_courant$
  fin si
fin pour
retourner *mot*

Exemple : A partir des mots circulants suivant $M_1 = 1, 6, 5, 6, 3, 4$ et $M_2 = 1, 2, 3, 2, 7$, l'algorithme calcule le mot M_{Fusion}.

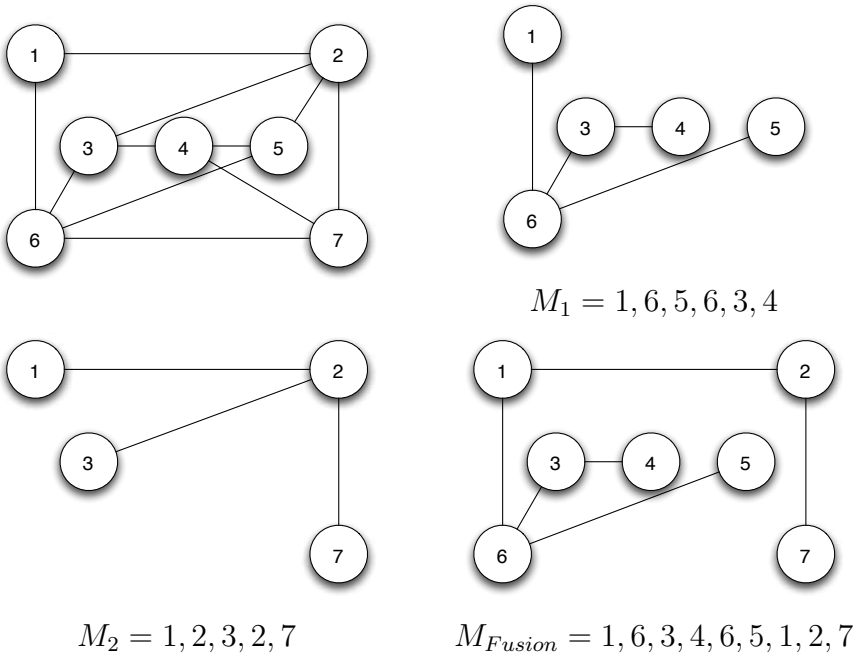


FIG. 5.4 – Exemple de fusion de deux mots : Réseau, Arbre mot 1, Arbre mot 2 et Arbre issu de la fusion des mots M_1 et M_2

Le cas des jetons multiples est donc résolu. Nous avons, en outre, mis en place un mécanisme qui accélère la construction via le mot circulant d'un arbre couvrant en fusionnant les informations topologiques de chacun de ces jetons.

5.2.4 Algorithme principal

Notre algorithme doit converger d'une configuration arbitraire vers une configuration où un unique jeton circule. Nous avons précisé que l'utilisation de marches aléatoires permet de pallier au dynamisme des réseaux.

Néanmoins afin de résoudre le problème d'interblocage de communication, nous avons utilisé des horloges de garde. Nous avons aussi montré que l'utilisation de ces horloges garantit la présence de jeton mais ne peut en aucun cas garantir l'unicité du jeton. Afin de résoudre ce problème, nous avons introduit la vague de réinitialisation, mécanisme qui permet d'éviter la création de jetons non nécessaires. Cette vague nécessite la création et la maintenance d'un arbre couvrant auto-adaptatif, ce que nous avons effectué via le mot circulant. Les reconfigurations topologiques peuvent donc engendrer une incohérence dans ce mot, c'est pourquoi il faudra utiliser le test interne (cf. Section 4.3) afin d'éliminer les incohérences contenues dans le mot du jeton.

Cela nous donne l'algorithme suivant :

Algorithme 22 Algorithme auto-stabilisant de circulation de jeton sur le site p

[**Réception d'un message (***Jeton***) provenant du site** *emmeteur*]
$ajout(p, Jeton.mot)$
$Test\_interne(Jeton.mot)$ */* Correction du mot reçu */*
si $File\_de\_messages\_non\_vide()$ **alors**
  pour tout $J \in$ file de messages **faire**
    / Cas d'une collision : plusieurs jetons sur le même site */*
    $Test\_interne(J.mot)$
    $Fusion\_jetons(Jeton, J)$
    $consomme\_message(J)$
  fin pour
fin si
$Jeton.saut \longleftarrow Jeton.saut + 1$ */* Incrémentation du nombre de saut du jeton */*
si $Jeton.saut = \max_{(i,j)} h(i,j) - \mathcal{N}$ **alors**
  / La vague de réinitialisation doit être déclenchée */*
  pour tout $v \in fils_p$ (d'après $Jeton.mot$) **faire**
    envoyer $reinit, Sous\_arbre(Jeton.mot, v))$ à v
  fin pour
  $Jeton.saut \longleftarrow 0$
fin si
Envoyer $Jeton$ à i choisi uniformément au hasard parmi $Vois_p$
Réinitialise l'horloge de garde à $\max_{(i,j)} h(i,j) \times (\Delta + \Theta)$
[**Déclenchement d'une horloge de garde**] */* Création d'un nouveau jeton */*
$Jeton.mot \longleftarrow \emptyset$
$Jeton.saut \longleftarrow 0$
Envoi $Jeton$ à i choisi uniformément au hasard parmi $Vois_p$
Réinitialiser l'horloge de garde à $\max_{\forall (i,j) \in V^2} h(i,j) \times (\Delta + \Theta)$
[**Réception d'un message (***reinit*, *mot***)**]
pour tout $v \in fils_p$ (d'après $Jeton.mot$) **faire**
  / Propagation de la vague */*
  envoyer $reinit, Sous\_arbre(Jeton.mot, v))$ à v
fin pour
Réinitialiser l'horloge de garde à $\max_{\forall (i,j) \in V^2} h(i,j) \times (\Delta + \Theta)$

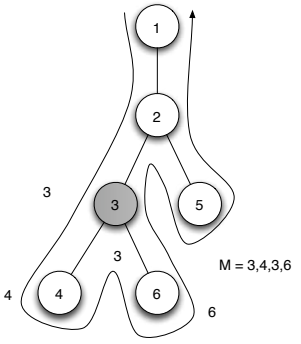
Cet algorithme utilise les fonctions suivantes :

- $File\_de\_message\_non\_vide$: retourne vrai si un autre message est présent sur le site. Il s'agit d'une *collision :* plusieurs jetons sont sur le même site.
- $Fusion\_jeton$: fusionne les jetons et les informations topologiques qu'ils contiennent. Le nouveau compteur de sauts est le maximum entre les deux jetons fusionnés (cf. Algorithme 20).
- $Test\_interne$: vérifie la cohérence du mot contenu dans le jeton par rapport au voisinage du site p. Si un nœud i est décrit comme fils de p dans le mot, mais n'appartient pas à $Vois_p$, le sous-arbre enraciné en i est effacé du mot (cf. Algorithme 4.3.1).

– *Sous_arbre*(v, m) : retourne le sous-arbre dont la racine est v à partir du mot m. L'algorithme est le suivant :

Algorithme 23 Procédure : Sous_arbre(m : mot, v : entier) retourner m : mot

$A \longleftarrow Construit\_arbre\_depuis\_mot(m)$
$cpt \longleftarrow 0$
tant que $cpt \neq \#fils_v$ (dans A) **faire**
  Parcourir l'arbre A par un parcours en profondeur d'abord en commençant par le nœud v
  pour chaque nœud visité dans A **faire**
    si $noeud\_courrant \notin mot$ **alors**
      $m \longleftarrow m + tmp + id\_noeud\_courrant$
      $tmp \longleftarrow \emptyset$
    sinon
      $tmp \longleftarrow id\_noeud\_courrant$
    fin si
    si $id\_noeud\_courrant = v$ **alors**
      $cpt \longleftarrow cpt + 1$
    fin si
  fin pour
fin tant que
retourner m



Soit le mot $M = 1, 2, 3, 4, 2, 5, 3, 6$. L'algorithme construit l'arbre A associé au mot M, visite chacun des sommets de A en suivant un parcours en profondeur d'abord. Une fois le sommet 3 visité une première fois, l'algorithme transcrit le mot correspondant en s'arrêtant dès lors que le parcours sort du sous-arbre enraciné en 3.

FIG. 5.5 – Exemple de calcul de sous-arbre enraciné sur le site 3

5.3 Preuves

Dans cette section, nous démontrons que notre algorithme réalise effectivement une circulation de jeton, malgré l'occurrence de fautes transitoires. Afin de prouver notre algorithme, nous combinons les résultats des marches aléatoires et le comportement de la vague de réinitialisation.

Les résultats des marches aléatoires montrent qu'à partir du moment où la rapidité de convergence de la marche aléatoire est inférieure au temps écoulé entre deux fautes transitoires, tout site est visité infiniment souvent par un jeton circulant aléatoirement. Ainsi la circulation d'un jeton unique est assurée.

Quelques rappels :

Propriété 5.1 (Couverture) *En partant d'un nœud arbitraire, un jeton circulant aléatoirement finira par visiter tous les nœuds du graphe de communication.*

Propriété 5.2 (Collision) *Si plusieurs jetons circulent sur le graphe, ils finiront par se renconter sur un nœud.*

Nous montrons maintenant que notre algorithme corrige les fautes transitoires. Nous avons utilisé l'auto-stabilisation comme paradigme pour concevoir un algorithme tolérant aux pannes transitoires.

Nous devons donc démontrer la propriété de convergence - notre algorithme converge vers un comportement correct (état légal) - la propriété de clôture - après avoir adopté un comportement correct, le système n'adoptera pas de comportement arbitraire - et enfin la propriété de correction - l'algorithme répond aux spécifications du problème.

Afin de prouver la stabilisation de notre algorithme, nous introduisons ici les notions qui nous permettrons de définir le comportement correct de notre algorithme.

Définition 5.1 *Un jeton est dit* complet*, si le mot associé au jeton contient l'ensemble des identités des sites qui sont présents dans le réseau.*

Définition 5.2 *Un jeton est dit* cohérent*, si le mot contenu au sein du jeton permet la construction d'un arbre (éventuellement sous-arbre) couvrant sur le réseau.*

Remarque 5.1 *A partir des Définitions 5.1, 5.2, un jeton complet et cohérent permet la construction d'un arbre couvrant sur le réseau.*

Définition 5.3 *Nous appelons un site* visité*, un site qui peut être atteint par la vague de réinitialisation (ce site a été visité par un jeton cohérent qui est toujours présent dans le réseau).*

Définition 5.4 *Predicat Etat Légal ($\mathcal{EL}$)*
Un état est dit légal *si les conditions suivantes sont vérifiées :*

- *Il y a* exactement un jeton cohérent et complet *dans le réseau d'interconnexion. (Predicat $\mathcal{JUCC}$)*
- *Tous les sites ont été visités. (Predicate $\mathcal{TSV}$)*

Lemme 5.1 *Un site visité est un site qui ne peut plus créer de nouveau jeton.*

Démonstration Un site visité est un site qui est atteignable par la vague de réinitialisation : il est visité par la vague de réinitialisation avant d'avoir pu créer un nouveau jeton.

□

Remarque 5.2 *Un site dont l'identité est stockée dans le mot circulant d'un jeton, n'est pas nécessairement un site visité : le mot peut être corrompu.*

Lemme 5.2 *Le nombre de sites visités augmente.*

Démonstration S'il n'y a pas de jeton, alors une horloge de garde finit par se déclencher et un jeton est créé. Sinon les jetons se déplacent et visitent de nouveaux sites. Ceux-ci peuvent être atteints par la vague de réinitialisation. Ces sites sont donc des sites visités.

□

Lemme 5.3 *Tous les sites finissent par devenir des sites visités : à partir d'une configuration arbitraire $\mathcal{C}$, nous avons $\mathcal{C} \longrightarrow \mathcal{TSV}$.*

Démonstration Par la Proposition 5.1 et l'itération du Lemme 5.2.

□

Remarque 5.3 *Tous les sites ont été visités, mais pas nécessairement par le même jeton. Les mots de certains de ces jetons peuvent même être incohérents. Mais les différentes vagues de réinitialisation propagées à travers les (possiblement incorects) arbres couvrants empêchent la création de tout nouveau jeton.*

A partir de cette étape, il existe au moins un jeton et aucune nouvelle création de jeton n'est possible.

Lemme 5.4 *Tous les jetons deviennent cohérents.*

Démonstration La procédure de test interne vérifie localement la cohérence du mot contenu dans le jeton et le corrige éventuellement. Quand le jeton a visité tous les sites, toutes les incohérences ont été supprimées du jeton, et le jeton est donc maintenant un jeton cohérent.

□

Lemme 5.5 *Il existe au moins un jeton qui a visité tous les sites.*

Démonstration Une marche aléatoire visite finalement tous les nœuds d'un graphe (Proposition 5.1).

□

Remarque 5.4 *Cette condition est suffisante mais pas obligatoirement nécessaire : si deux jetons fusionnent, la vague de réinitialisation du jeton ainsi produit atteint tous les sites qui ont été visités par ces deux jetons. Ainsi, une vague de réinitialisation peut atteindre tous les sites, même si le jeton associé n'a pas visité tous les sites.*

Lemme 5.6 *Pour toutes configuration satisfaisant* $\mathcal{C} \in \mathcal{TSV}, \mathcal{C} \longrightarrow \mathcal{JUCC}$

Démonstration L'algorithme fusionne les jetons s'ils se rencontrent sur le même site au même moment. [IJ90, TW91] ont montré qu'un tel mécanisme, finit par fusionner tous les jetons en un seul. Enfin, la fusion de jetons préserve les informations topologiques, donc le jeton issu de cette fusion contient un arbre couvrant sur le réseau. Nous avons donc un unique jeton complet et cohérent.

□

Théorème 5.1

$$\forall \mathcal{C}, \mathcal{C} \longrightarrow \mathcal{EL}$$

Démonstration Par les lemmes précédants, nous avons à partir d'une configuration arbitraire $\mathcal{C}, \mathcal{C} \longrightarrow \mathcal{TSV}$, d'une configuration $\mathcal{C} \in \mathcal{TSV}, \mathcal{C} \longrightarrow \mathcal{JUCC}$ et $\mathcal{EL} = \mathcal{TSV} \cap \mathcal{JUCC}$.

□

Ainsi, d'une configuration arbitraire, notre algorithme converge vers un état légal et satisfait la circulation de jeton auto-stabilisante.

5.4 Performances

Nous présentons ici les performances de notre algorithme par rapport à trois aspects. Le premier des aspects, c'est le temps moyen qu'un site doit attendre entre deux visites successives du jeton. Le second de ces aspects, c'est le temps de convergence de notre algorithme partant d'une configuration arbitraire pour atteindre une configuration légale. Enfin, le troisième aspect concerne le coût des opérations que nous avons dû ajouter pour obtenir un algorithme auto-stabilisant.

5.4.1 Temps d'attente

Dans [IKOY02], les auteurs proposent une méthode permettant d'égaliser les temps d'attente (temps pour un site pour obtenir le jeton) sur chacun des sites. Le temps d'attente ($h(i,i)$) étant inversement proportionnel aux probabilités en distribution stationnaire, leur méthode consiste à déterminer les valeurs des arêtes pour égaliser les probabilités à l'état stationnaire. Ces valeurs sont donc dépendantes de la topologie du réseau. Dans notre cas (environnement dynamique et distribué), il est impossible d'adapter instantanément ces valeurs. De plus une adaptation étape par étape des valuations de chacune des arêtes, risque de surcharger le réseau sans apporter un résultat convaincant : si le dynamisme du réseau est trop important, les valeurs des arêtes sont sans cesse mises à jour sans converger vers les valeurs qui permettent d'atteindre l'équité. Néanmoins, si la topologie du réseau est "presque" régulière (c'est-à-dire que les degrés de chacun des sites sont très proches par rapport au nombre total d'arêtes), notre algorithme est proche de l'équité car le temps moyen d'attente est égale à $\frac{2m}{\deg(i)}$. Sur un anneau par exemple, le temps moyen d'attente est de n. Nous avons effectué des simulations sur les temps d'attente (cf. Section 6.4)

5.4.2 Temps de convergence

Nous présentons ici l'analyse du temps de convergence. Le temps moyen de convergence est difficile à analyser : d'une configuration arbitraire, la convergence est assurée, mais la manière d'atteindre l'état légal n'est pas unique. Nous décomposons la phase de convergence en plusieurs étapes et indiquons le temps moyen pour franchir chacune de ces étapes.

- Le système converge d'une configuration sans jeton vers une configuration où il existe au moins un jeton. Ceci est effectué en $Tmax = \max_{(i,j)\in V^2} h(i,j) \times (\Delta + \Theta)$.
- Le système converge d'une configuration où plusieurs jetons sont présents vers une configuration où seul un jeton est présent. La collision de tous les jetons s'effectue en $M = 8n^3/27 \times (\Delta + \Theta)$ (se référer à [TW91] pour plus de détails sur le temps de rencontre entre différentes marches aléatoires).
- Le système doit corriger un jeton contenant un mot incohérent. Le temps moyen pour effacer toutes les incohérences d'un mot est le temps nécessaire au jeton contenant le mot, de visiter tous les sommets du réseau (de manière que chaque site puisse procéder au test interne), donc du même ordre que le temps de couverture $C \times (\Delta + \Theta)$.
- Le mot du jeton doit être complet pour éviter les créations inopportunes de jetons. Ceci est effectué quand le jeton a parcouru tous les nœuds du réseau ($C \times (\Delta + \Theta)$).

où $Tmax$ est le délai écoulé avant qu'une horloge se déclenche ($T_{max} = \max_{(i,j)\in V^2} h(i,j)$), C est le temps de couverture d'une marche aléatoire sur un graphe et M est le temps de rencontre entre plusieurs marches aléatoires. Nous avons développé une méthode dans le Chapitre 3 qui permet le calcul automatique des grandeurs caractéristiques d'une marche aléatoire.

Exemple : Soit une configuration où il y a un seul jeton (j), mais le mot qu'il contient est incohérent. Il est donc tout à fait possible que d'autres jetons soient recréés (la vague de réinitialisation ne peut pas atteindre tous les sites). Une fois que le jeton j aura parcouru tous les sites du réseau (C), le mot de j devient alors cohérent mais aussi complet (l'élimination des erreurs se fait conjointement avec la reconstruction de l'arbre). Il n'y aura alors plus de création de nouveaux jetons. L'algorithme fusionne alors tous les jetons en un seul (la fusion des informations topologiques n'entraîne pas de création d'un mot incohérent si les mots fusionnés sont cohérents) lors des collisions des jetons (M). Dans ce cas, le temps de stabilisation est de $(M + C) \times (\Delta + \Theta)$.

5.4.3 Coût de la stabilisation

En terme de messages, le surcoût de la stabilisation représente uniquement le coût de la vague de réinitialisation : $n-1$ messages propagés tous les $Tmax - \mathcal{N}$ sauts du jeton. Le surcoût de cette vague est donc de $\frac{n-1}{Tmax-\mathcal{N}}$ par saut de jeton. Or il est possible d'initialiser les horloges à n'importe quelles valeurs $Tmax$ pourvu qu'elles soient d'une part égales sur tous les sites et d'autre part supérieures à $\mathcal{N}$. Nous avons choisi dans notre algorithme $Tmax = \max_{(i,j)\in V^2} h(i,j)$ puisqu'il s'agit du temps moyen de retour du jeton, mais il est possible de modifier ces valeurs pourvu qu'elles soient supérieures à $\mathcal{N}$. Augmenter la valeur de $Tmax$ va réduire le nombre de vagues propagées. Néanmoins, cela augmente

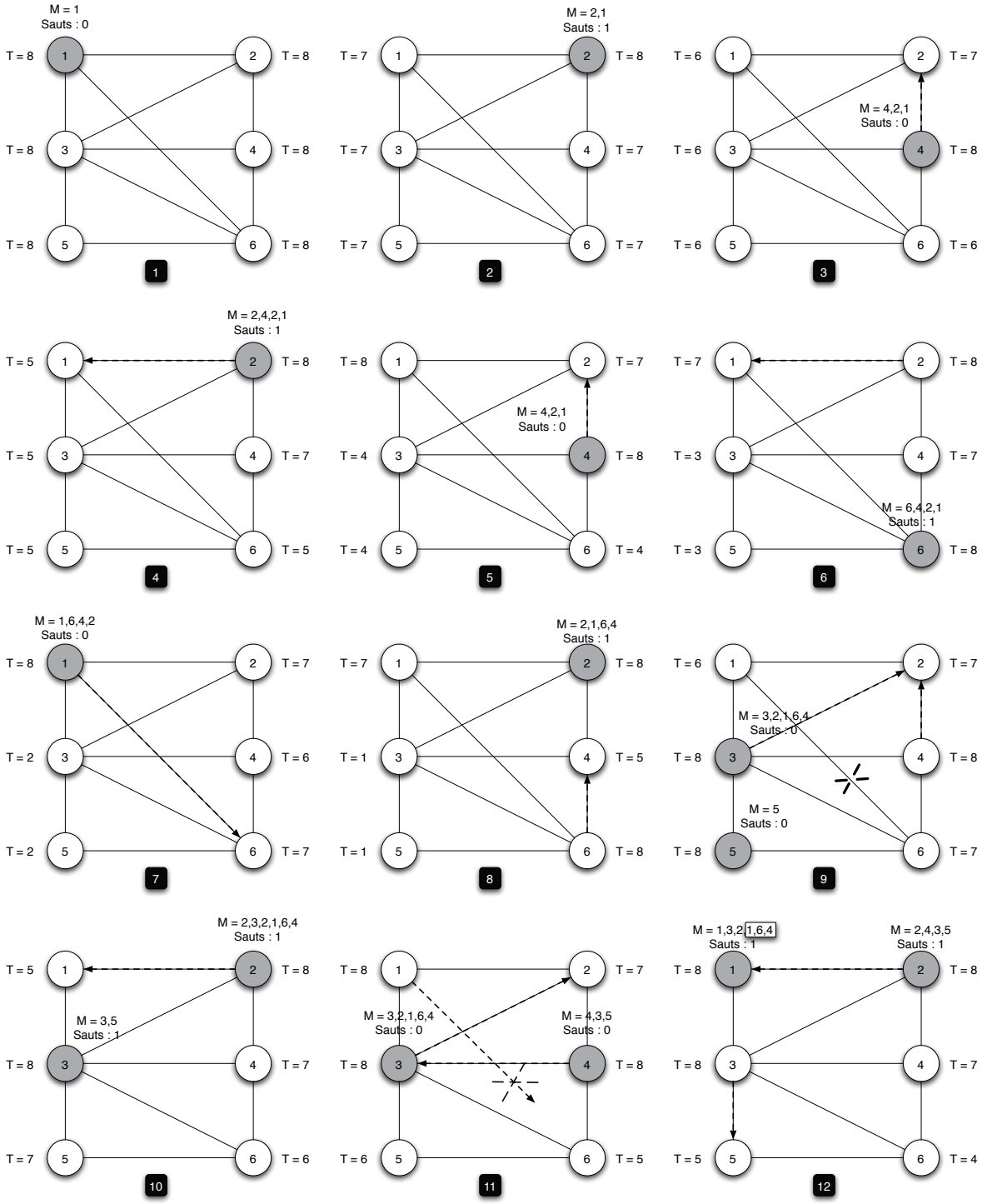
aussi le temps de convergence : S'il n'y a plus de jeton, plus $Tmax$ est grand plus il faut attendre avant la création d'au moins un nouveau jeton.

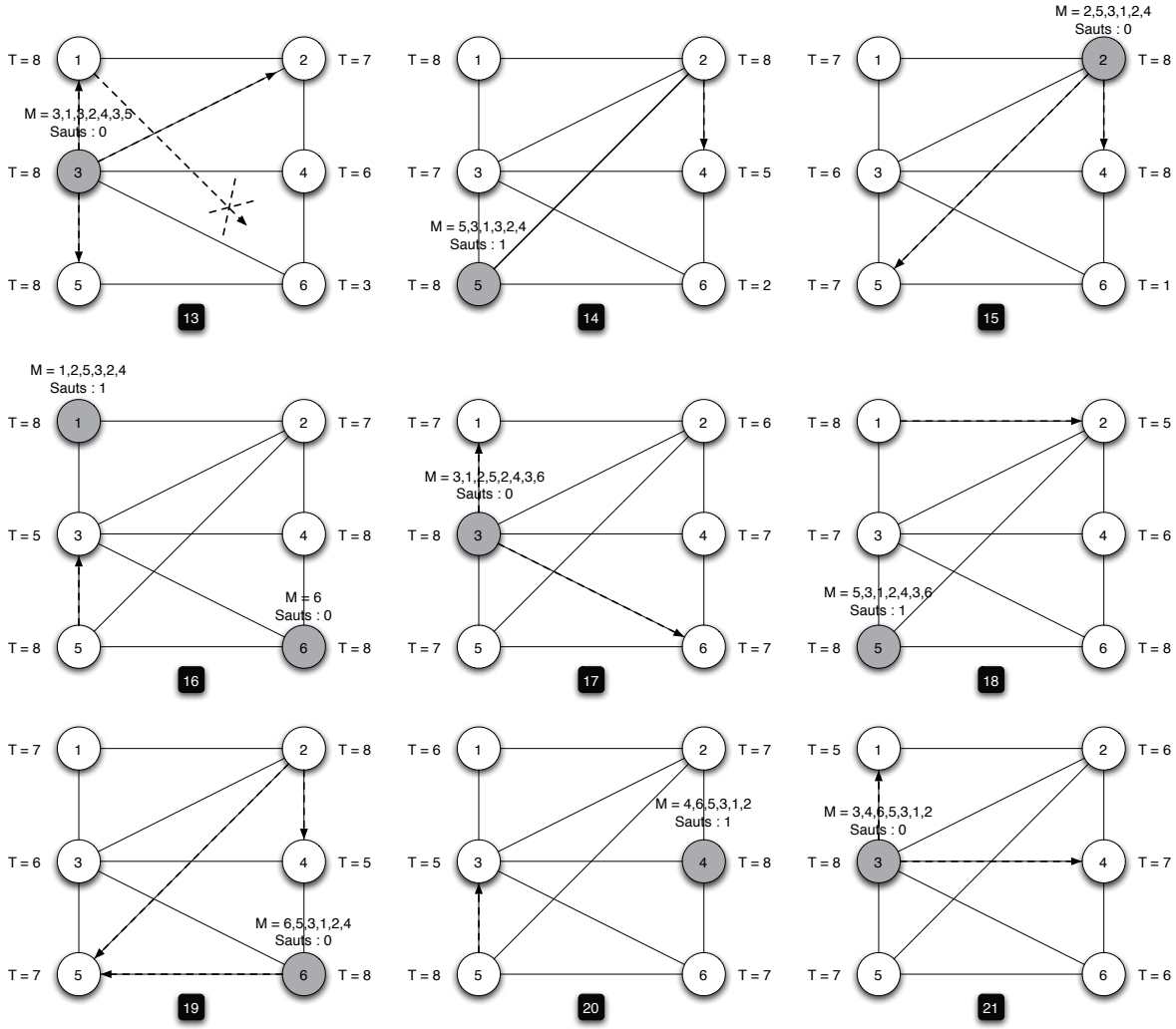
En terme de place, la taille du jeton est plus importante que dans une version non tolérante aux fautes (dans une version non tolérante aux pannes, le recours à la vague de réinitialisation est inutile et le jeton peut alors être vide). Dans notre cas, nous stockons plusieurs informations : un mot circulant permettant la création de l'arbre et le nombre de sauts qu'a effectué le jeton. Cela nous donne une complexité en $O(2n)$. Il est possible de réduire la complexité de notre mot circulant en le remplaçant par la représentation d'arbre par table que nous avons introduite en Section 4.3.2. La complexité serait alors en $O(n)$.

Un autre aspect, la tolérance aux pannes de notre algorithme est assurée dans un système synchrone. La version non tolérante aux pannes fonctionne parfaitement dans un système asynchrone.

5.5 Exemple détaillé d'exécution

Nous détaillons ici un exmple d'exécution de notre algorithme sur un réseau d'au maximum 6 sites et avec une valeur $Tmax = 8$. Les vignettes représentent les configurations successives du système.


M = 1
Sauts : 0
M = 2,1
Sauts : 1
M = 4,2,1
Sauts : 0
M = 2,4,2,1
Sauts : 1
M = 4,2,1
Sauts : 0
M = 6,4,2,1
Sauts : 1
M = 1,6,4,2
Sauts : 0
M = 2,1,6,4
Sauts : 1
M = 3,2,1,6,4
Sauts : 0
M = 5
Sauts : 0
M = 2,3,2,1,6,4
Sauts : 1
M = 3,5
Sauts : 1
M = 3,2,1,6,4
Sauts : 0
M = 4,3,5
Sauts : 0
M = 1,3,2,1,6,4
Sauts : 1
M = 2,4,3,5
Sauts : 1



Etape 1 : Le jeton est sur le site 1, toutes les horloges sont initialisées à $Tmax = 8$.

Etape 2 : Le jeton se déplace vers le site 2, le mot contenu ainsi que le nombre de sauts sont mis à jour. Toutes les horloges ont été décrémentées (exceptée celle du site 2 : le jeton est sur ce site).

Etape 3 : A l'arrivée du jeton sur le site 4, la condition de déclenchement de la vague est vérifiée. Celle-ci est donc propagée à travers l'arbre construit par le mot 4,2,1. Le nombre de sauts du jeton est donc replacé à 0. Toutes les horloges sont décrémentées (exceptée celle de 4).

Etape 4 : Le site 2 reçoit un message de réinitialisation qu'il propage vers le site 1. Il réinitialise son horloge et reçoit le jeton, effectue sa mis-à-jour (mot et sauts). Toutes les horloges sont décrémentées (exceptée celle de 2)

Etape 5 : Le jeton revient en 4 et est mis à jour. Une nouvelle vague est déclenchée sur l'arbre 4-2-1. Le site 1 reçoit le message de réinitialisation de la précédente vague, il réinitialise donc son horloge à $Tmax$. Les autres horloges (2,3,5,6) sont décrémentées.

Etape 6 : Le jeton se déplace vers le site 6 qui le met à jour. Le site 2 reçoit un message

de réinitialisation, il transmet celui-ci vers le site 1 et réinitialise son horloge. Les autres horloges (1,3,4,5) sont décrémentées.

Etape 7 : Le site 1 reçoit la vague et le jeton (MàJ), il réinitialise son horloge. Une nouvelle vague est déclenchée sur l'arbre 1-6-4-2. Toutes les horloges sont décrémentées (exceptée celle de 1).

Etape 8 : Le site 6 reçoit la vague, il réinitialise son horloge, et propage la vague vers le site 4. Le site 2 reçoit le jeton et le met à jour. Les autres horloges (1,3,4,5) sont décrémentées.

Etape 9 : Le jeton arrive sur le site 3 et est mis à jour. **Un jeton est créé sur le site 5, car l'horloge du site 5 a atteint la valeur 0 (elle est alors réinitialisée).** Une nouvelle vague est déclenchée sur le site 3 à travers l'arbre 3-2-1-6-4. **Le canal 1-6 devient indisponible. Le mot du premier jeton est donc incohérent.** Le site 4 reçoit un message de réinitialisation de la précédente vague, il réinitialise son horloge et propage ce message vers le site 2. Les horloges des sites 1,2 et 6 sont décrémentées.

Etape 10 : Le premier jeton arrive sur le site 2 et le deuxième sur le site 3. Ils sont mis à jour. Le site 2 reçoit un message de réinitialisation, il réinitialise son horloge et propage ce message vers le site 1. Les horloges des sites 1,4,5 et 6 sont décrémentées.

Etape 11 : Les deux jetons se déplacent respectivement vers les sites 3 et 4 et sont mis à jour. Ils doivent tous les deux propager une vague de réinitialisation respectivement sur les arbres 3-2-1-6-4 et 4-3-5. Le site 1 reçoit un message de réinitialisation de la précédente vague. Il tente de retransmettre ce message vers le site 6. **Ce message est perdu car il n'y a plus de lien entre 1 et 6**. Le site 1 réinitialise néanmoins son horloge. Les horloges des sites 2,5 et 6 sont décrémentées.

Etape 12 : Les deux jetons se déplacent respectivement vers les sites 1 et 2 et sont mis à jour. Le site 2 reçoit un message de réinitialisation qu'il propage vers le site 1, idem pour le site 3 vers le site 5. Ils réinitialisent donc leur horloge. **Le site 1 détecte une incohérence dans le mot qu'il possède : le site 6 est décrit comme étant son fils mais le site 6 n'apparaît pas dans $Vois_1$. Il coupe alors le mot, il reste 1,2,3.** Les sites 3 et 2 recoivent un message de réinitialisation. Ils sont propagés respectivement vers 5 et 1. Les horloges des sites 4,5 et 6 sont décrémentées.

Etape 13 : Les deux jetons arrivent sur le site 3. **Il y a collision, les mots sont fusionnés.** Une vague de réinitialisation est déclenchée sur le site 3 à travers l'arbre 3-2-4-3-1-3-5. Le site 5 reçoit un message de réinitialisation et réinitialise son horloge. Le site 1 reçoit un message de réinitialisation qu'il propage vers le site 6. **Ce message est perdu car il n'y a plus de lien entre 1 et 6**. Le site 1 réinitialise néanmoins son horloge. Les horloges des sites 2,4 et 6 sont décrémentées.

Etape 14 : Le jeton se déplace vers le site 5. **Un nouveau lien se forme entre les sites 5 et 2.** Les sites 1,2 et 5 reçoivent un message de réinitialisation, le site 2 propage celui-ci vers le site 4. Les horloges des sites 3,5 et 6 sont décrémentées.

Etape 15 : Le jeton arrive sur le site 2, est mis à jour et déclenche une nouvelle vague à travers l'arbre 2-5-3-1-2-4. Le site 4 reçoit un message de réinitialisation. Les horloges des sites 1,3,5 et 6 sont décrémentées.

Etape 16 : Le jeton arrive sur le site 1. Les sites 4 et 5 reçoivent un message de réinitialisation. Le site 5 propage celui-ci vers le site 3. **Un nouveau jeton est créé sur le site 6.** Les horloges des sites 2 et 3 sont décrémentées.

Etape 17 : les deux jetons arrivent sur le site 3 : collision, les jetons fusionnent et le jeton produit déclenche une vague sur l'arbre 3-1-2-5-2-4-3-6 (le site 3 recevant en outre un message de réinitialisation, il propagera alors 2 messages de réinitialisation vers 1). Les horloges des sites 1,2,4,5 et 6 sont décrémentées. **Le système est maintenant dans une configuration légale.**

Etape 18 : Le jeton se déplace vers le site 5. Le site 1 reçoit un message de réinitialisation qu'il propage vers le site 2. Les horloges des sites 2, 3, 4 et 6 sont décrémentées.

Etape 19 : Le jeton se déplace vers le site 6 où il déclenche une nouvelle vague sur l'arbre 6-5-3-1-2-4. Le site 2 reçoit un message de réinitialisation qu'il propage vers les sites 4 et 5. Les horloges des sites 1, 3, 4 et 5 sont décrémentées.

Etape 20 : Le jeton se déplace vers le site 4 qui reçoit aussi un message de réinitialisation. Le site 5 en reçoit 2, il en propage un vers le site 3. Les horloges des sites 1, 2, 3 et 6 sont décrémentées.

Etape 21 : Le site 3 reçoit le jeton et un message de réinitialisation. Il déclenche aussi une nouvelle vague sur l'arbre 3-4-5-3-1-2. Les horloges des sites 1, 2, 4, 5 et 6 sont décrémentées.

...

Dans cet exemple, la vague de réinitialisation est propagée tous les 2 tours. Cela est assez élevé, mais permet, dans le cas d'une perte de jeton, de pouvoir en créer un nouveau de manière rapide.

Nous avons montré ici à l'aide d'un exemple comment notre algorithme s'adaptait aux reconfigurations topologiques (vignettes 9 et 14). Nous n'avons pu mettre en évidence toutes les combinaisons de fautes transitoires qui peuvent survenir, néanmoins nous avons dans cet exemple pris garde à montrer le fonctionnement de chacun des mécanismes que nous avons utilisés.

5.6 Conclusion

Nous avons, dans ce chapitre, présenté le problème de circulation de jeton tolérante aux pannes. Notre problématique a été de concevoir une solution décentralisée fonctionnant sur un réseau arbitraire, ce qui nous a naturellement conduit à l'utilisation d'une marche aléatoire. Nous avons été amenés à proposer l'utilisation d'une procédure décentralisée d'horloges de garde permettant dans le cas où l'on suspecte la perte d'un jeton, de produire un nouveau jeton. Nous avons aussi montré que l'utilisation conjointe d'horloges de garde et d'une marche aléatoire, ne permet pas de garantir la propriété de clôture de notre algorithme. C'est pourquoi nous avons mis en place un autre mécanisme : la vague de réinitialisation. Ce mécanisme est déclenché à l'initiative du jeton et permet d'empêcher la création d'un nouveau jeton. Cette vague est propagée au travers d'un arbre adaptatif stocké dans le jeton sous forme de mot circulant. Après avoir prouvé notre algorithme, nous

avons donné la mesure théorique des performances de notre algorithme. Ces performances sont exprimées en fonction des valeurs caractéristiques des marches aléatoires (qui peuvent être calculées à partir de la méthode que nous proposons au chapitre 3). Nous avons conclu en présentant un exemple assez conséquent de l'exécution de notre algorithme qui permet de mieux appréhender le fonctionnement général de celui-ci.

Chapitre 6

Application aux réseaux ad-hoc et résultats expérimentaux

Résumé : *Nous présentons un algorithme d'allocation de ressources tolérant aux pannes pour les réseaux ad-hoc. Cet algorithme repose sur les principes présentés dans les chapitres précédents. La conception d'algorithmes pour les réseaux ad-hoc nécessite souvent des hypothèses sur le schéma de mobilité des périphériques. Le caractère auto-stabilisant de notre algorithme est vérifié si la mobilité du réseau respecte une condition que nous avons mis en évidence. Ces travaux ont fait l'objet d'une publication dans une conférence internationale [BBFNar]. Nous avons effectué des expérimentations pour évaluer les performances générales de notre algorithme, mais aussi des grandeurs caractéristiques d'une marche aléatoire.*

6.1 Introduction

Dans ce chapitre, nous proposons une solution auto-stabilisante au problème de la k-exclusion dans les réseaux ad-hoc.

Le problème auquel nous nous intéressons, la k-exclusion, [FLBB89] est une généralisation du problème d'exclusion mutuelle : au plus k processeurs sont autorisés à exécuter leur code de section critique simultanément. Notre solution est tolérante aux fautes transitoires et adaptative aux reconfigurations topologiques sous l'hypothèse d'une contrainte de mobilité souple (contrainte que nous mettons en évidence Section 6.3). w Comme le précise [Ray91b], deux types de solutions sont envisageables pour résoudre ce problème : les solutions fondées sur les permissions et les solutions fondées sur les jetons (cf. Section 1.4.4). Nous utilisons une solution basée sur la circulation de k jetons distincts telle que nous l'avons décrite dans le chapitre précédant [BV95, WCM01, HV01]. Un processeur en possession d'un des k jetons, peut entrer en section critique (SC). Une telle solution est bien adaptée aux réseaux ad-hoc [WCM01].

Dans [WWV01], les auteurs utilisent une solution basée sur un jeton pour résoudre l'exclusion mutuelle dans un réseau ad-hoc. Leur algorithme maintient un graphe orienté acyclique avec une destination dynamique. Tous les processeurs sont totalement ordonnés et le processeur dont l'identité est la plus petite, est toujours le possesseur du jeton. Pour

entrer en section critique, un processeur construit un graphe orienté acyclique jusqu'au possesseur du jeton. Pour résoudre le problème de k-exclusion, dans un réseau ad-hoc, les auteurs de [WCM01] proposent une généralisation de l'algorithme précédent [WWV01]. Néanmoins, ces algorithmes ne sont pas auto-stabilisants.

Dans [CW02, CW05], les auteurs proposent une solution auto-stabilisante au problème de 1 exclusion sur un réseau ad-hoc. Leur solution fonctionne dans un réseau qui est structuré en anneau dynamique virtuel grâce à une circulation de jeton (Algorithme $\mathcal{LRV}$). Le jeton est périodiquement généré par un processeur distingué. Leur algorithme nécessite que le réseau soit statique pendant que l'algorithme converge. Dans [KY04], les auteurs présentent une circulation de jeton déterministe adaptée aux réseaux ad-hoc. Leur algorithme est auto-stabilisant mais aussi robuste face aux reconfigurations topologiques sous une hypothèse modérée de mobilité.

Contrairement aux solutions tolérantes aux pannes, notre solution ne nécessite pas la maintenance d'une structure virtuelle. C'est pourquoi, tous les processeurs ont la possibilité de créer de nouveaux jetons. La maintenance d'une structure virtuelle peut être coûteuse du fait de "l'empilement de protocole" qu'elle nécessite. Néanmoins, pour assurer son caractère tolérant aux pannes, notre solution requiert, comme les autres solutions, des hypothèses sur la mobilité des nœuds.

Nous explicitons d'abord notre solution. Les algorithmes de circulation de jeton qui sont présentés ici, utilisent une table comme représentation d'un arbre couvrant (cf. Section 4.3.2). Nous présentons alors dans un schéma de preuve, la condition pour laquelle notre algorithme conserve son caractère auto-stabilisant. Enfin, nous fournissons un ensemble de résultats expérimentaux sur les marches aléatoires et sur les performances qu'offre notre solution.

6.2 Allocation de ressources dans les réseaux ad-hoc

Le problème de k-exclusion est caractérisé par les propriétés suivantes [FLBB89] :
- **Sûreté :** Au plus k processeurs peuvent exécuter simultanément leur code de section critique.
- **Vivacité :** Chaque nœud requérant l'accès à sa section critique finira par l'obtenir en un temps fini.

Dans notre solution, chacun des k jetons circule perpétuellement. La possession d'un jeton autorisant le nœud à entrer en section critique, il y au plus k nœuds qui peuvent entrer en section critique. La propriété de sûreté est donc vérifiée. Chaque jeton réussit à visiter tous les nœuds en un temps fini à la condition que les reconfigurations topologiques le permettent (c'est-à-dire que le graphe reste connexe et que la mobilité des nœuds ne soit pas guidée par un adversaire qui choisit les arêtes à déplacer afin de "contrer" les visites du jeton [PSM+04]). Ainsi, la propriété de vivacité est satisfaite.

6.2.1 Fonctionnement de l'algorithme

Pour concevoir une solution auto-stabilisante au problème de k-exclusion dans les réseaux ad-hoc, nous devons considérer une configuration initiale arbitraire et nous devons

faire converger le système vers son comportement correct, c'est-à-dire vers une configuration où exactement k jetons circulent dans le réseau. Nous utilisons k jetons distincts auxquels sont associées k vagues de réinitialisation. $\mathcal{K}$ désigne l'ensemble des identités correctes que peut avoir un jeton. Nous avons donc $|\mathcal{K}| = k$ et donc chaque nœud possède k horloges permettant la régénération de jetons valides.

Dans un souci de clarté, nous distinguons trois phases qui caractérisent le comportement de l'algorithme. Ces phases peuvent se dérouler de manière simultanée.

- **La phase préliminaire** consiste à éliminer les jetons comportant une identité corrompue. C'est-à-dire qu'un jeton J tel que $J.id \notin \mathcal{K}$ est ignoré. [Algorithme 24 : ligne 4]. Cette opération est réalisée en $O(1)$.
- **La phase de production de jeton** Cette phase doit produire suffisamment de jetons : au moins k jetons distincts. Ceci est effectué par les k horloges de garde positionnées sur chaque nœud. Elles sont initialisées à $h(i, j)$.
- **La phase de fusion** Cette phase élimine les jetons dupliqués [Algorithme 24 : ligne 10]. Dans [TW91], il est montré que cela peut être réalisé en $O(n^3)$.

L'algorithme assure qu'en temps fini, une configuration avec exactement k jetons distincts est atteinte. Chaque nœud possèdant un jeton avec une identité valide peut entrer en section critique. Sous une contrainte souple de mobilité (cf. Section 6.3), le problème de la k exclusion tolérante aux pannes est résolu. Cette condition de mobilité est relative au fonctionnement de la vague de réinitialisation : si les différentes vagues ne peuvent pas atteindre tous les nœuds du réseau, alors il est possible qu'un ou plusieurs de ces nœuds créent des copies de jetons déjà existants.

Notre algorithme utilise une procédure décentralisée pour la régénération des jetons : k horloges de garde positionnées sur chaque site (cf. [Algorithme 24 : ligne 32–41]). Ainsi, chaque site a indistinctement la possibilité de produire des nouveaux jetons valides. Nous présentons maintenant notre solution. Au sein de chaque jeton, la maintenance de l'arbre couvrant adaptatif est effectuée avec la représentation sous forme de table (cf. Section 4.3.2).

Nous détaillons les différentes phases de notre solution sur un exemple (Figure 6.1). Nous avons pour cet exemple $k = 2$ (les identités sont spécifiées à l'aide de couleurs) et pour chaque site, deux horloges de garde (une par identité de jeton).

1. Nous sommes dans une configuration arbitraire avec deux jetons sur les sites 2 et 3. Le jeton sur le site 2 a une identité corrompue, il est donc éliminé du réseau. Il n'y a plus qu'un seul jeton dans le système.
2. Après 2 étapes, 2 jetons noirs sont créés sur les sites 3 et 6. Le jeton gris a transité vers 2 puis 4. Nous avons maintenant toutes les identités de $\mathcal{K}$ représentées sur le réseau.
3. Les trois jetons circulent et finalement après 3 étapes, les deux jetons noirs se rencontrent sur le site 2. Celui-ci les fusionne en un seul. Nous avons maintenant exactement 2 jetons distincts d'identités non corrompues.
4. Ces deux jetons circulent et peuvent éventuellement se rencontrer.

Nous n'avons pas explicité le fonctionnement de la vague de réinitialisation afin de ne pas surcharger les schémas. Celle-ci est propagée périodiquement par un jeton j, pour réinitialiser les horloges correspondant à j et donc éviter la production de copie de j.

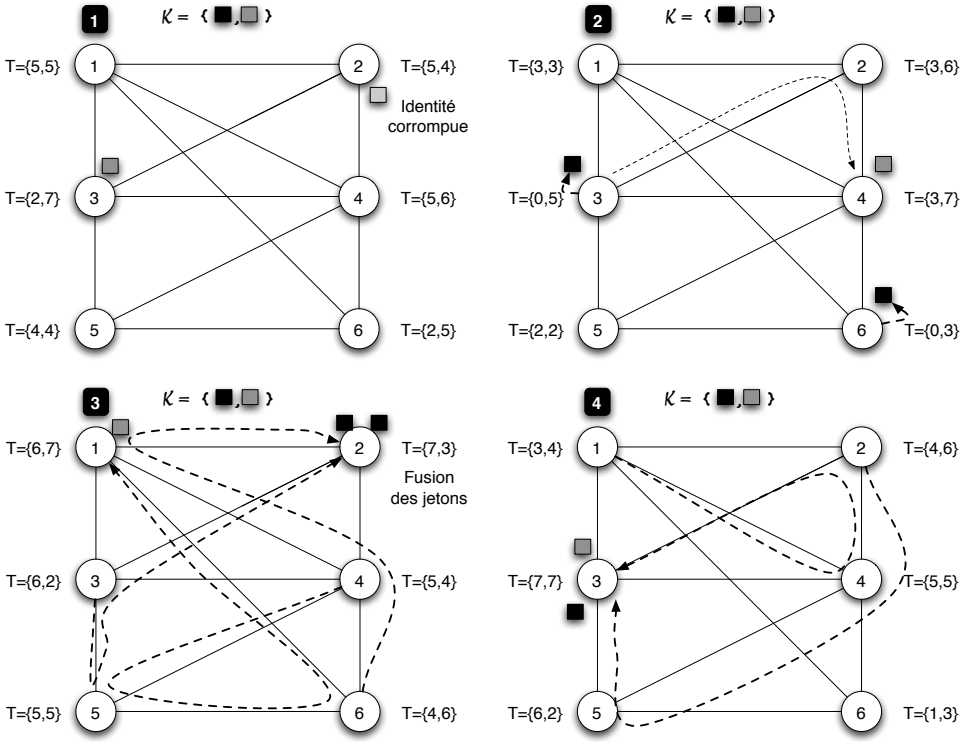


FIG. 6.1 – Illustration des trois phases de notre algorithme

6.2.2 L'algorithme

Chaque jeton contient :
- un champ identifiant, *J.id*. Un jeton avec une identité valide est un jeton J tel que $J.id \in \mathcal{K}$.
- un champ compteur, *J.saut*. Ce champ représente le nombre d'arêtes qu'a parcouru le jeton au cours de sa traversée.
- un champ table, *J.table*. *J.table* est la table qui représente l'arbre adaptatif (cf. Section 4.3.2) qui permet la propagation de la vague de réinitialisation. A chaque déplacement du jeton, l'arbre contenu sous forme de table est mis à jour.

Le calcul de l'arbre couvrant est effectué en mettant à jour la structure de données stockée dans le jeton durant son parcours [Algorithme 24 : ligne 7–8]. Les fautes transitoires peuvent altérer le contenu du jeton, ou bien les reconfigurations topologiques peuvent entraîner l'incohérence d'un jeton. Nous avons résolu ce problème dans le chapitre 4 : chaque site à la réception d'un jeton teste localement sa cohérence et éventuellement corrige celui-ci (cf. Algorithme 11). En un temps fini, le jeton visite tous les nœuds du réseau et donc toutes les incohérences sont effacées.

La propagation de la vague est déclenchée par l'[Algorithme 24 : ligne 20–24] ou diffusée par l'[Algorithme 24 : ligne 42–47]. La procédure *Sous_arbre*() calcule le sous-arbre restant sur lequel la vague doit être diffusée [Algorithme 26], et la procédure *Fusion_jetons*() permet de fusionner les informations topologiques [Algorithme 25].

Algorithme 24 Algorithme auto-stabilisant de k-exclusion sur le nœud p

1: [**Réception d'un message (***Jeton***) depuis le nœud** *emetteur*]
2: **si** $Jeton.id \notin \mathcal{K}$ **alors**
3: ignorer *Jeton /* Le jeton n'est pas valide */*
4: **fin si**
5: $Jeton.table[p] \longleftarrow p$*/* Mise-à-jour des informations topologiques de l'arbre maintenu */*
6: $Jeton.table[emetteur] \longleftarrow p$
7: **si** $File\_de\_messages\_non\_vide()$ **alors**
8: **pour** Chaque jeton J dans la file de message **faire**
9: **si** $Jeton.id = J.id$ **alors**
10: $Fusion\_jetons(Jeton, J)$*/* La fusion n'est effectuée qu'entre jeton de même identifiant */*
11: $Consomme\_message(J)$
12: **fin si**
13: **fin pour**
14: **fin si**
15: $Test\_interne(Jeton.table)$
16: $Jeton.saut \longleftarrow Jeton.saut + 1$
17: **si** $Jeton.saut \times (\Delta + \Theta) \geq (\max\{h(i,j)/(i,j) \in V^2\} - n \times (\Delta + \Theta)$ **alors**
18: */* Propagation de la vague de réinitialisation */*
19: **pour tout** v fils de p (conformément à $Jeton.table$) **faire**
20: Envoyer $(reinit, Sous\_arbre(Jeton.table, v), Jeton.id)$ à v
21: **fin pour**
22: $Jeton.saut \longleftarrow 0$
23: **fin si**
24: **si** Section Critique Requise **alors**
25: **Entrer en section critique**
26: **fin si**
27: Envoyer $(Jeton)$ à i choisi uniformément au hasard parmi $Vois_p$
28: $horloge[id] \longleftarrow \max\{h_{ij}/(i,j) \in V^2\} \times (Delta + \Theta)$
29: [**Déclenchement de** $horloge[h]$] */* Une création de jeton h survient */*
30: **pour** $i = 0$ à $n - 1$ **faire**
31: $Jeton.table[i] \longleftarrow nondefini$
32: **fin pour**
33: $Jeton.id \longleftarrow h$
34: $Jeton.table[p] \longleftarrow p$
35: $Jeton.hop \longleftarrow 0$
36: Envoyer $(Jeton)$ à i choisi uniformément au hasard parmi $Vois_p$
37: $horloge[h] \longleftarrow \max\{h_{ij}/(i,j) \in V^2\} \times (\Delta + \Theta)$
38: [**Réception d'un message (***reinit, table, id***)**] */* Réinitialise* $horloge[id]$ **/*
39: **pour tout** v fils de p (conformément à $table$) **faire**
40: Envoyer $(reinit, sous\_arbre(table, v), id)$ à v
41: **fin pour**
42: $horloge[id] \longleftarrow \max\{h_{ij}/(i,j) \in V^2\} \times (\Delta + \Theta)$

Algorithme 25 Procédure : Fusion_jetons(J, Jt : Jeton)

$J.saut = max(J.saut, Jt.saut)$
$Jt.table[p] \longleftarrow p$
$Jt.table[emetteur] \longleftarrow p$
pour $k = 0$ à $n - 1$ **faire**
  si $(J.table[k] = undefined) \cap (Jt.table[k] \neq undefined)$ **alors**
    $J.table[k] \longleftarrow Jt.table[k]$
  fin si
fin pour

Algorithme 26 Procédure : Sous_arbre(t : tableau, v : entier) retourne t' : tableau

pour tout $i \in V$ **faire**
  $t'[i] \longleftarrow undefined$
fin pour
$t'[v] \longleftarrow v$
$liste \longleftarrow \{v\}$
pour tout $h \in liste$ **faire**
  pour $j = 0$ à $n - 1$ **faire**
    si $t[j] = h$ **alors**
      $t'[j] = h$
      $liste \longleftarrow liste \cup \{j\}$
    fin si
  fin pour
  $liste \longleftarrow liste - \{h\}$
fin pour
retourner t'

6.3 Schéma de preuves

Partant d'une configuration initiale arbitraire, nous montrons que, sous une contrainte de mobilité souple, le système atteint une configuration avec exactement k jetons.

Définition 6.1 *Un jeton J est appelé* non corrompu *si $J.id \in \mathcal{K}$.*

Définition 6.2 *Un jeton T est appelé* cohérent *si $J.table$ est la représentation d'un arbre couvrant.*

Lemme 6.1 *D'une configuration initiale arbitraire l, le système atteindra une configuration satisfaisant la condition S_C : "il n'y a pas de jeton corrompu dans le système".*

Démonstration [Algorithme 24 : line 2–4], un jeton corrompu est ignoré par tout nœud le recevant.

□

Lemme 6.2 *Seuls des jetons non corrompus (ie $J.id \in \mathcal{K}$) peuvent être crées.*

Lemme 6.3 *Soit $\mathcal{L}$ l'ensemble des identités des jetons dans le réseau. S'il existe une identité i telle que $i \in \mathcal{K} \backslash \mathcal{L}$, alors un jeton J avec $J.id = i$ sera créé.*

Démonstration Considérons une identité $i \in \mathcal{K} \backslash \mathcal{L}$. Il n'y a pas de jeton J tel que $J.id = i$ dans le réseau et aucune vague de réinitialisation associée à un tel jeton. Un nœud a finalement donc une de ses horloges (la i-ème) qui arrive à expiration et produit une nouvelle occurrence du jeton i (par le Lemme 6.2 avec une nouvelle identité [Algorithme 24 : ligne 32–41]).

□

Remarque 6.1 *Aussi longtemps qu'un jeton i n'a pas visité tous les nœuds du réseau, une copie du jeton i peut être créée.*

Lemme 6.4 *Partant d'un configuration satisfaisant S_C, l'algorithme atteindra une configuration satisfaisant la condition S_A : "au moins k jetons distincts circulent dans le système".*

Démonstration Soit $\mathcal{L}$ l'ensemble des identités des jetons présents dans le système et l le nombre de jetons circulants dans le système. Nous avons les deux cas suivants :

Cas 1 $0 \leq l < k$. Il y a moins de k jetons dans le système (avec possiblement des jetons dupliqués).

Cas 2 $l > k$. Il y a plus de k jetons dans le système. Certains d'entre eux peuvent être dupliqués. Ce cas peut être divisé en deux sous-cas. (i) Pour tout $i \in \mathcal{K}$ il existe un jeton J avec $J.id = i$ et (ii) il y a moins de k jetons distincts dans le réseau.

Finalement par le Lemme 6.3, de nouveaux jetons sont créés et pour tout $i \in \mathcal{K}$ il existe au moins un jeton J tel que $J.id = i$ dans le réseau.

□

Si le réseau est statique nous avons montré, Chapitre 5, qu'un jeton finit par contenir l'image d'un arbre couvrant. Les reconfigurations topologiques peuvent survenir pendant la construction des arbres couvrants. Les changements topologiques comme l'ajout d'arêtes ou bien la mobilité des arêtes externes aux arbres sont possibles... La condition suivante CMS est suffisante et même un peu restrictive :

Propriété 6.1 (CMS1) *"Une arête choisie par le jeton pour devenir arête de l'arbre couvrant adaptatif (ie traversée par le jeton) doit rester statique."*

Puisque le jeton circule aléatoirement, une arête donnée au temps t candidate pour appartenir à l'arbre couvrant, peut finalement ne plus appartenir à l'arbre après quelque temps. Finalement, notre contrainte de mobilité oblige $n - 1$ arêtes par arbre à rester statiques, mais pas nécessairement en même temps.

Si la condition $CMS1$ est vérifiée, une fois que chaque jeton a visité l'ensemble des nœuds du réseau, nous avons :

Lemme 6.5 *A la condition $CMS1$ vérifiée, chaque jeton J devient cohérent.*

Lemme 6.6 *D'une configuration satisfaisant S_A, l'algorithme atteint une configuration satisfaisant S_E , c'est-à-dire qu'il n'y a plus de création de nouveaux jetons.*

Démonstration Par le Lemme 6.5, chaque nœud est atteignable par la vague de réinitialisation associée à chaque jeton. Un nœud visité par un jeton J ne pourra plus déclencher son horloge $J.id$ (en effet celle-ci n'atteint jamais la valeur 0), et donc la création de nouveaux jetons est impossible.

□

Lemme 6.7 *Si la condition $CMS1$ est vérifiée, d'une configuration satisfaisant S_E, l'algorithme atteint une configuration S_L satisfaisant le prédicat suivant : "Il y a exactement k jetons distincts dans le réseau".*

Démonstration Une fois que chacun des jetons (au moins k avec exactement k identités distinctes) a visité tous les nœuds du réseau, par le Lemme 6.6, il ne peut plus y avoir création de jeton avec une nouvelle identité ou avec une identité dupliquée. Par la Propriété de collision (Section 2.5.2) les jetons dupliqués fusionnent. Les collisions entre jetons peuvent se produire à tout moment et en particulier pendant la phase de création. Au final, il n'y a plus de jetons dupliqués et k jetons distincts.

□

Théorème 6.1 *L'algorithme est auto-stabilisant si la condition $CMS1$ est vérifiée.*

Démonstration La propriété de convergence est montrée par les Lemmes 6.1, 6.4, 6.6 et 6.7. Concernant la propriété de clôture, il suffit de remarquer que dans une configuration satisfaisant S_L, chaque nœud est visité par les jetons ou atteint par une vague de réinitialisation avant qu'une de ses horloges se déclenche. Ainsi, la création de nouveaux jetons est impossible.

□

Une fois l'état stable atteint, la contrainte de mobilité peut être relaxée. La mobilité des arêtes externes aux arbres couvrants ou les ajouts d'arêtes sont permis. Les arêtes des arbres peuvent aussi se déconnecter après la propagation de la vague de réinitialisation. Cette condition garantit le bon fonctionnement de nos vagues de réinitialisation. La contrainte devient alors :

Propriété 6.2 (CMS2) *Une vague de réinitialisation doit pouvoir être propagée sur l'arbre adaptatif maintenu au sein du jeton associé.*

Il y a alors exactement k jetons dans le réseau, la propriété de sûreté est vérifiée. Chaque jeton visite infiniment souvent tous les nœuds du réseau, la propriété de vivacité est aussi vérifiée.

Théorème 6.2 *L'algorithme est une solution auto-stabilisante au problème de la k exclusion si la condition $CMS2$ est vérifiée.*

6.4 Simulations et résultats

La simulation d'un système permet d'observer son comportement global et ses performances avant l'implantation en cas réels. Dans le cas d'algorithmes distribués, la première étape consiste à modéliser le support sur lequel est sensé s'exécuter l'algorithme. Dans [Rak94], l'auteur précise qu'il existe deux types de simulations :

La simulation de systèmes continus. Un système est modélisé sous forme d'équations différentielles qui régissent l'évolution du système. Nous trouvons dans ce cas, les simulations des marchés économiques, les simulations écologiques (modèles proie-prédateur),...

La simulation de systèmes à événements discrets . La modélisation de tels systèmes correspond à des règles de succession d'événements. L'évolution du système dépend alors des événements déjà survenus et de l'ordre temporel de ceux-ci.

Dans notre cas, il s'agit alors de simulations à événements discrets : chacun des sites consomme et produit des événements.

Le recours à des simulations est nécessaire lorsque :

1. Il n'est pas possible de prédire le comportement qu'adopte le système.

2. Le coût de test du système en conditions réelles est trop prohibitif. On procède alors à des simulations pour effectuer les ajustements à priori.

Les simulations permettent aussi de mieux appréhender les performances de l'algorithme simulé et éventuellement de proposer des conjectures. Néanmoins, elles ne permettent pas de calculer de bornes et ne démontrent pas non plus un résultat théorique. Les résultats de simulations n'ont qu'une valeur indicative et ne permettent pas de prouver une propriété.

La problématique de la simulation d'algorithmes distribués s'oriente en général vers le choix du modèle : celui-ci doit être suffisamment proche de la réalité pour prendre en compte toute la diversité des exécutions réalisables, mais ce modèle doit aussi être "léger" pour obtenir un temps de calcul acceptable et donc la réalisation d'un nombre suffisant d'exécutions. De plus, les systèmes devenant de plus en plus grands, il convient alors d'utiliser un simulateur bien adapté aux besoins et aux modèles de la simulation.

Nous avons simulé le comportement de marches aléatoires sur des réseaux afin de vérifier les performances théoriques montrées dans le Chapitre 3 mais aussi de proposer des conjectures sur le temps moyen d'attente pour obtenir la section critique avec notre algorithme de k-exclusion. Nos simulations ont été écrites en C++ à l'aide de la bibliothèque de simulation développée dans [Rab05]. Nous avons effectué nos simulations dans un cadre synchrone, c'est-à-dire à l'aide d'une horloge globale. Le protocole de simulation est le suivant : à chaque pulsation de l'horloge, tous les événements en suspens sont consommés et produisent de nouveaux événements qui seront consommés à la prochaine pulsation de l'horloge globale. Dans notre cas, les événements sont des réceptions et des émissions de jetons. Nous avons donc effectué des mesures expérimentales sur les temps de couverture, les temps de rencontre et les temps d'attente.

6.4.1 Temps de couverture

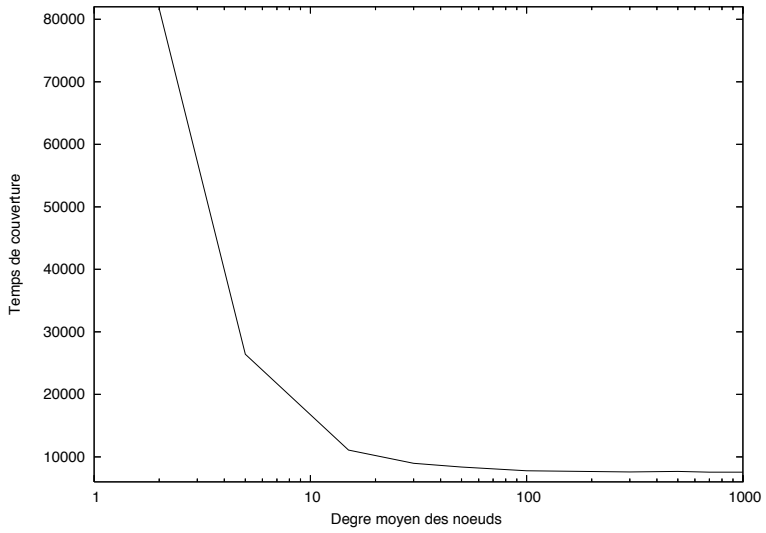
Nous avons tenté dans ces simulations de voir quels sont les paramètres qui sont les plus influents sur le temps de couverture d'une (ou plusieurs) marche(s) aléatoire(s). Nous avons donc considéré trois paramètres :

Le degré moyen d'un noeud : intuitivement plus le degré moyen par nœud est élevé, moins la marche aléatoire risque de visiter des nœuds déjà visités.

Le nombre de noeud du réseau : plus il y a de nœuds dans le réseau, plus le temps nécessaire à une marche aléatoire pour visiter tous les nœuds du réseau est long.

Le nombre de jeton circulant dans le réseau : intuitivement plus il y a de marches aléatoires, plus le temps nécessaire pour visiter tous les nœuds du réseau est court.

Pour chacun des paramètres, nous avons effectué 1000 simulations. Nous fournissons ici les valeurs moyennes.



Deg	2	5	15	30	50
C_{sim}	81670.9	26418	11084.7	8968.92	8371.39
Deg	100	300	500	700	1000
C_{sim}	7779.85	7685.16	7666.52	7559.14	7545.05

FIG. 6.2 – Influence du degré moyen sur le temps de couverture

Nous avons effectué ces simulations sur un graphe de 1000 nœuds avec un seul jeton. Le degré moyen semble n'avoir une influence que si celui-ci est petit. Sur notre simulation, si le degré moyen est supérieur à 3%, les résultats attendus sont de l'ordre de ceux que l'on obtient avec un graphe complet. Il apparaît donc que le degré moyen du réseau n'est un paramètre critique que si celui-ci est vraiment très faible par rapport au nombre de nœuds.

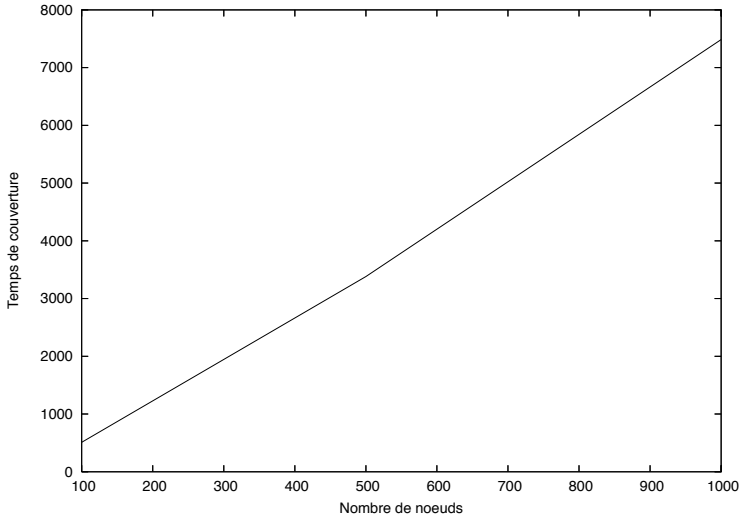


FIG. 6.3 – Influence du nombre de nœuds sur le temps de couverture

Nous avons effectué ces simulations sur un graphe complet avec un seul jeton. Les résultats que nous avons obtenus sont en adéquation avec les résultats présentés Chapitre 3 sur le graphe complet : c'est-à-dire une croissance quasi linéaire avec le nombre de nœuds du réseau, C est donc en $O(n)$.

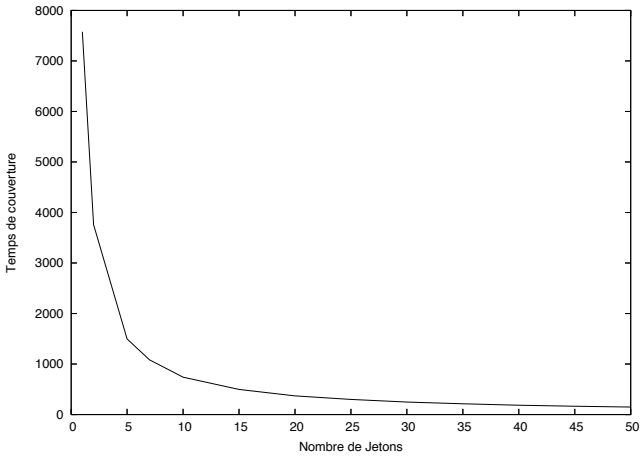


FIG. 6.4 – Influence du nombre de jetons sur le temps de couverture

Nous avons effectué ces simulations sur un graphe complet de 1000 noeuds. Il ne s'agit pas ici de la définition classique du temps de couverture : comme précisé Section 2.5.2, le

temps de couverture ne concerne qu'une seule marche aléatoire. Ce que nous avons simulé ici c'est le temps nécessaire à plusieurs marches aléatoires pour visiter tous les nœuds du réseau. Intuitivement, nous conjecturions que plus il y a de jetons, plus court est le temps nécessaire à la visite de tous les nœuds. Les résultats obtenus nous confirment cette intuition. Mieux, il semblerait qu'il existe une relation inversement proportionnelle entre le nombre de jetons et le temps de couverture :

Nb de jetons	1	2	5	7	10	15
C_{sim}	7577.15	3757.05	1497.09	1082.74	740.394	497.786
20	25	30	35	40	45	50
370.95	299.618	247.574	214.38	185.482	164.644	149.054

6.4.2 Temps de rencontre

Nous avons considéré dans ces simulations quatre paramètres qui nous apparaissaient importants pour le calcul du temps de rencontre :

Le degré moyen du graphe : plus le degré est haut plus les jetons risquent de se rencontrer rapidement.

Le nombre de nœuds du graphe : plus celui-ci est important moins les jetons risquent de se rencontrer.

Le nombre total de jeton : il s'agit ici du nombre total de jetons circulant aléatoirement sur le réseau.

Le nombre de jetons qui se rencontrent Il s'agit ici du nombre de jetons se rencontrant au même moment sur un même site. A priori une rencontre entre deux jetons est plus facile à obtenir qu'une rencontre entre 3 jetons.

Pour chacun des paramètres nous avons effectué 1000 simulations. Nous fournissons ici les valeurs moyennes.

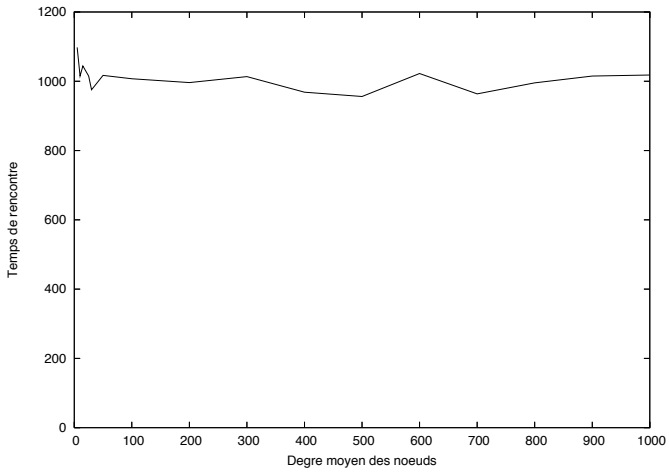


FIG. 6.5 – Influence du degré moyen sur le temps de rencontre

Nous avons effectué ces simulations sur un graphe de 1000 noeuds avec deux jetons se rencontrant parmi un nombre total de deux jetons. Il nous apparaissait évident que le degré avait une influence sur le temps de rencontre. Il n'en est rien : excepté pour des degrés extrêmement faibles, la valeur des temps de rencontre semble être constante. Pour des degrés moyens extrêmement faibles, nous risquons d'obtenir un graphe biparti, avec un jeton dans chacune des deux sous parties : dans ce cas, il n'y a jamais de rencontres entre les jetons.

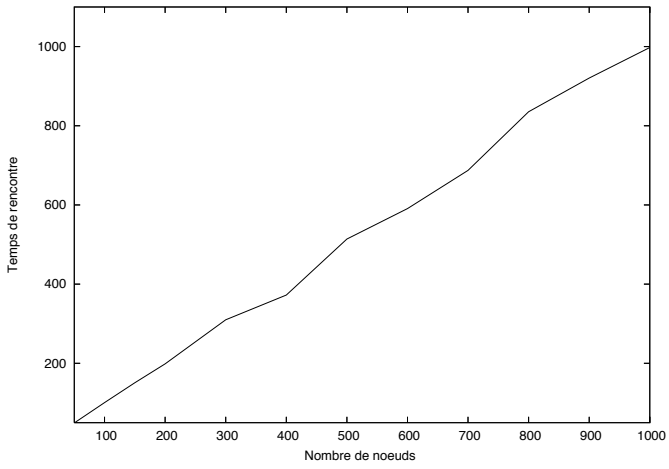
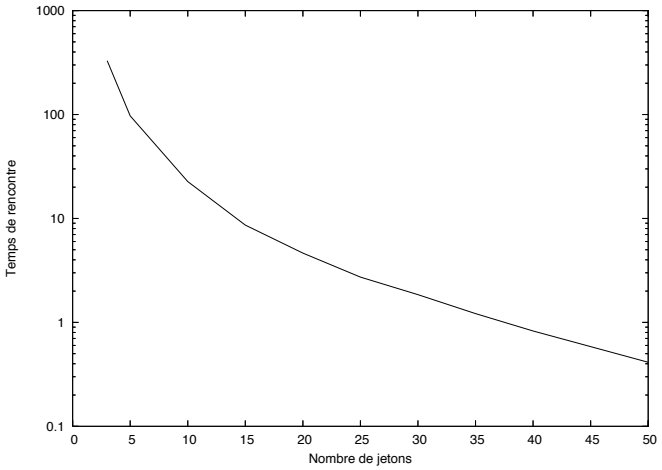


FIG. 6.6 – Influence du nombre de nœuds sur le temps de rencontre

Nous avons effectué ces simulations sur un graphe complet avec deux jetons se rencontrant parmi deux jetons. La relation entre le nombre de nœuds et le temps de rencontre semble être une relation linéaire. Nous savons ([TW91]) que le temps de rencontre est en $O(n^3)$ en général. Nous avons effectué nos simulations sur des graphes complets avec une stratégie de déplacement synchrome ce qui pourrait expliquer que le temps de rencontre dans nos simulations soit en $O(n)$.



Nb total de jetons	2	3	5	10	15
M_{sim}	1018.18	329.105	97.142	22.656	8.63
20	25	30	35	40	50
4.638	2.729	1.847	1.214	0.826	0.413

FIG. 6.7 – Influence du nombre total de jetons sur le temps de rencontre

Nous avons effectué ces simulations sur un graphe complet de 1000 noeuds avec des rencontres entre deux jetons. Nos simulations confirment le fait que plus il y a de jetons plus facilement deux jetons peuvent se rencontrer.

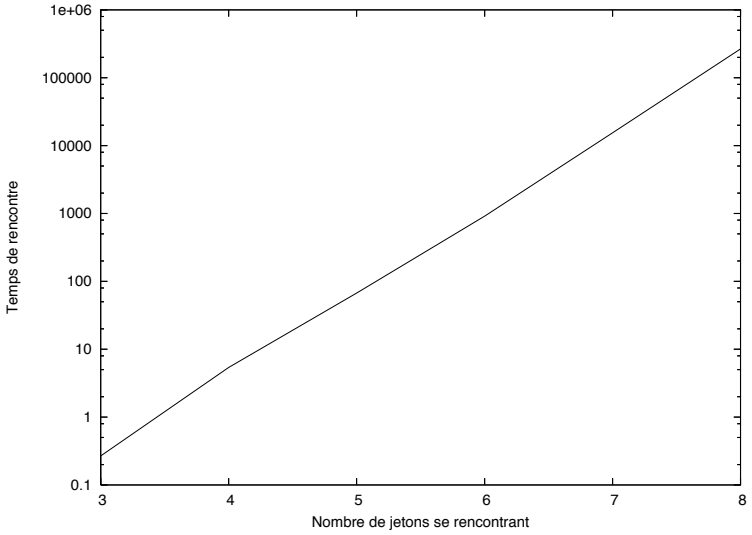


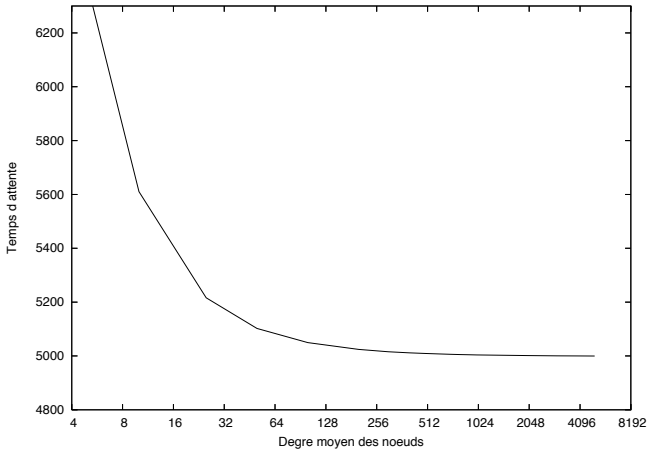
FIG. 6.8 – Influence du nombre de jetons se rencontrant sur le temps de rencontre

Nous avons effectué ces simulations sur un graphe complet de 100 noeuds avec 50 jetons. Conformément à notre intuition, le temps de rencontre croit de manière très importante si le nombre de jetons se rencontrant est important. Nous envisagions de calculer les temps de rencontres jusqu'à des rencontres de 25 jetons mais nous avons dû stopper nos simulations, au vu du temps de calcul requis pour des valeurs nettement inférieures. Il semblerait qu'il s'agisse ici d'un accroissement exponentiel. Il semble donc qu'utiliser des rencontres accidentelles d'un nombre trop important de jetons affecte profondément les performances des algorithmes. Dans le cadre de nos travaux nous avons conçu des algorithmes exploitant des rencontres accidentelles de deux jetons. Les performances ne sont donc pas affectées.

6.4.3 Temps d'attente

Nous allons dans le cadre de l'algorithme fourni dans ce chapitre, fournir des résultats expérimentaux sur les temps moyens d'attente avant de pouvoir entrer en section critique. Il s'agit du temps écoulé entre deux visites successives de jetons sur un même nœud. Nous avons effectué ces simulations sur plusieurs types de réseau : des réseaux statiques afin de voir comment agissent les paramètres *classiques*, mais aussi des réseaux dynamiques afin de voir comment notre algorithme réagit face à la mobilité des nœuds. Nous avons dans le cadre des réseaux statiques observé deux paramètres : le degré moyen et le nombre total de jetons dans le réseau. Pour le cas des réseaux dynamiques, nous avons simulé le modèle de mobilité basé sur une marche aléatoire (cf. Section 2.2).

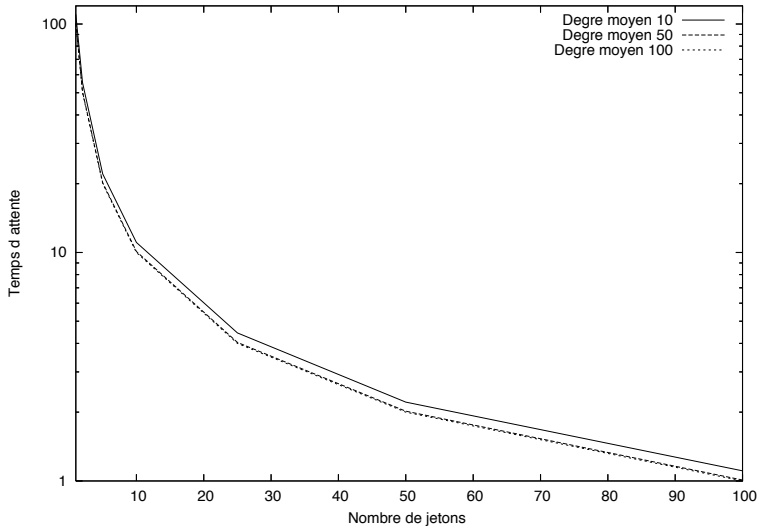
Cas des réseaux statiques



Degrés moyen	5	10	25	50	100	200
$h(i,i)_{sim}$	6368.03	5610.35	5216.20	5102.25	5049.82	5024.33
	300	400	500	600	700	800
	5015.57	5011.64	5009.12	5007.49	5006.14	5005.06
	900	1000	2000	3000	4000	5000
	5004.62	5003.99	5001.43	5000.61	5000.27	4999.87

FIG. 6.9 – Influence du degré moyen sur le temps d'attente

Nous avons effectué ces simulations sur un réseau de 5000 nœuds avec un seul jeton. Nous remarquons qu'il existe une faible relation entre le temps d'attente et le degré moyen, ce qui confirme l'analyse que nous proposons Section 5.4 : le temps d'attente dans le cas où il n'y a qu'un seul jeton c'est le temps de retour, c'est-à-dire $h(i,i)$. Comme nous l'avons précisé Section 2.5.3, le temps de retour est égal à $\frac{2m}{\deg(i)}$, il est donc fonction du degré moyen : m est le nombre d'arêtes et $\deg(i)$ est le degré de i, ces deux paramètres sont fonction du degré moyen.



Nombre de jetons	1	2	5	10	25	50	100
Temps d'attente degré 10	110.20	55.34	22.06	11.05	4.44	2.21	1.10
Temps d'attente degré 50	100.98	50.49	20.20	10.09	4.03	2.02	1.01
Temps d'attente degré 100	99.99	49.99	20.00	9.99	3.99	2.00	1.00

FIG. 6.10 – Influence du nombre de jetons sur le temps d'attente

Nous avons effectué ces simulations sur un réseau de 100 nœuds avec successivement des degrés moyens de 10, 50 et 100 et en faisant varier le nombre de jetons de 1 à 100. Nous constatons que le temps d'attente est inversement proportionnel au nombre de jetons, c'est-à-dire que par simulation nous vérifions la conjecture suivante : le temps d'attente est égal à $\frac{h(i,i)}{nbjeton}$. L'influence des degrés ici est assez faible. En effet nous n'avons pas choisi les valeurs des degrés moyens qui perturbent les performances de notre algorithme. Ici le degré moyen est compris entre 10 et 100, il est donc toujours supérieur à 10% du nombre total de nœuds. Or la Figure 6.9 nous montre que pour un degré suffisant (supérieur à 5% du nombre de nœuds) les performances de notre algorithme semblent ne pas être affectées.

Cas des réseaux dynamiques : modèle de mobilité basé sur une marche aléatoire

Nous avons aussi évalué par simulation les performances de notre algorithme dans un environnement mobile. Nous présentons donc ici les résultats que nous avons obtenu pour le modèle de mobilité basé sur des marches aléatoires dans les conditions suivantes : l'espace de simulation est un cube de 30 mètres de coté, il y a une population de 100 périphériques mobiles et chacun de ceux-ci a un rayon de transmission de 5 mètres.

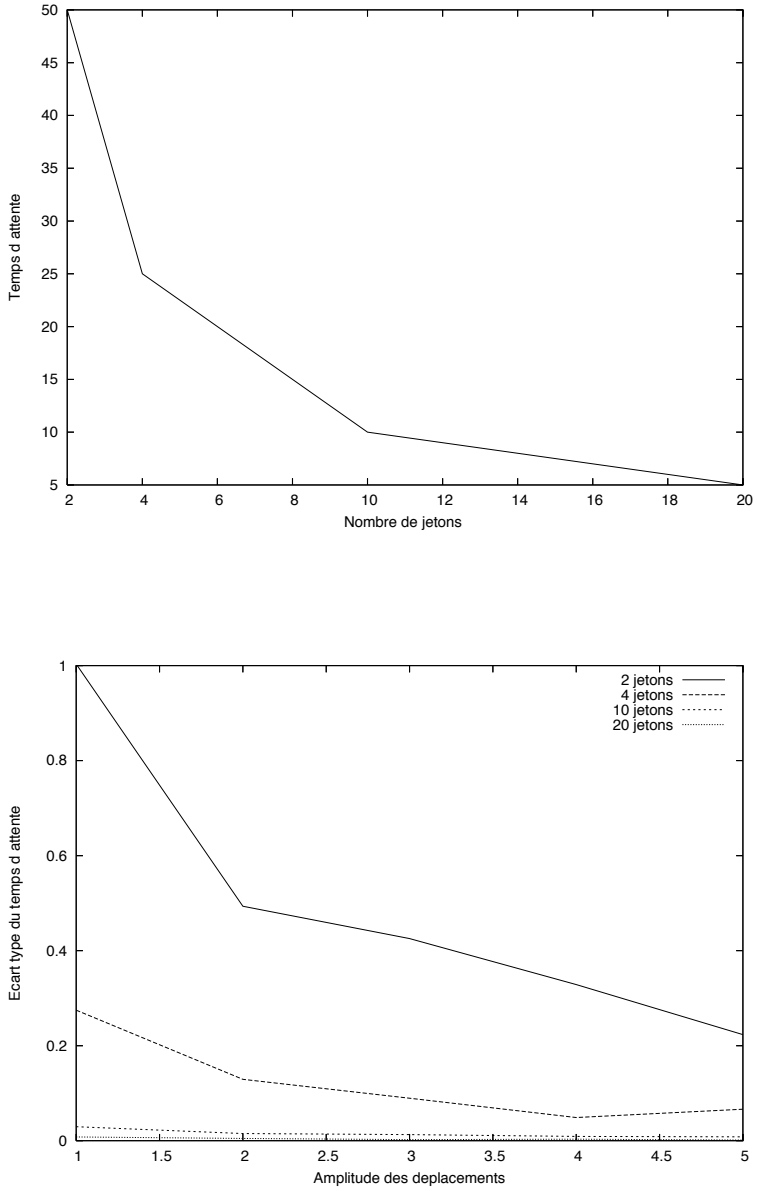


FIG. 6.11 – Temps d'attente et écart type sur un modèle de mobilité basé sur des marches aléatoires

Dans ce modèle de mobilité, chaque périphérique se déplace aléatoirement sans suivre de trajectoire prédéfinie. Nous avons fait varier l'amplitude des déplacements ainsi que le nombre de jeton. Notre première constatation concerne l'amplitude des déplacements : celle-ci n'affecte quasiment pas les performances de notre algorithme. En effet les résultats obtenus concordent parfaitement avec les résultats que nous avons obtenus dans le cadre de réseaux statiques (cf. Figure 6.10). Le temps d'attente est égal à $\frac{h(i,i)}{nbjeton}$. Par contre la

mobilité semble affecter l'écart type : plus la mobilité a une grande amplitude, plus l'écart type est petit. Il semble donc que la mobilité des réseaux contrebalance l'aléa généré par notre politique de circulation aléatoire.

6.5 Conclusion

Nous avons présenté dans ce chapitre une solution auto-stabilisante au problème d'allocation de ressources basée sur la circulation de jetons auto-stabilisante que nous avons décrite au chapitre précédant. Notre solution contrairement aux autres solutions, n'utilise pas de structures virtuelles construites au dessus du réseau d'interconnexion, ce qui permet dans le cas des réseaux ad-hoc une plus grande flexibilité : la mobilité des nœuds n'entrave pas la circulation de jeton mais uniquement le caractère auto-stabilisant de notre algorithme (possiblité de création de nouveaux jetons inutiles). Néanmoins, nous avons déterminé un critère de mobilité (Propriétés 6.1 et 6.2) qui permet de garantir les propriétés de convergence et de clôture de notre algorithme.

Nous avons proposé une évaluation expérimentale des performances de notre algorithme ainsi que des grandeurs caractéristiques qui permettent une approximation du temps de convergence. Les résultats de ces simulations montrent que l'utilisation de marches aléatoires ne peut être aussi performant que des solutions déterministes quand la topologie du graphe d'interconnexion est connue. Néanmoins, les performances qu'elles affichent, restent pertinentes dès lors que la topologie n'est pas connue et particulièrement dans le cas des réseaux mobiles où leur topologie est en évolution permanente.

Conclusion et Perspectives

Notre méthode de calcul des grandeurs caractéristiques des marches aléatoires, nous a permis de valider l'utilisation des marches aléatoires dans les réseaux de tailles raisonnables : globalement les complexités affichées sont de l'ordre de $O(n \log n)$ en moyenne et de $O(n^3)$ dans le pire des cas. Nous avons expliqué l'intérêt que proposait l'approche de conception d'algorithmes basés sur les marches aléatoires : adaptativité à des topologies arbitraires et résistance aux reconfigurations topologiques du réseau d'interconnexion. Les algorithmes que nous avons écrits sont en effet basés sur les trois propriétés fondamentales des marches aléatoires : percussion, couverture et collision.

Le mot circulant nous a permis de construire et d'adapter une structure couvrante tolérante aux pannes au sein même du jeton, fournissant à celui-ci un support de communication pour mener à bien les tâches qui lui incombe. Nous avons proposé deux types de structures, l'arbre couvrant dans le cas des réseaux non orientés et l'anneau dans le cas des réseaux orientés. Ces structures topologiques ne sont pas fractionnées et réparties sur l'ensemble des sommets mais elles sont centralisées au sein même du jeton permettant au site visité d'avoir temporairement une vision globale du réseau. Nous nous sommes servis de ce support dans le cadre des réseaux ad-hoc, pour l'écriture d'une solution au problème d'allocation de ressources.

Nous avons utilisé l'arbre adaptatif maintenu dans le mot circulant pour effectuer une diffusion d'information. Nous avons en particulier introduit le mécanisme de vague de réinitialisation : il permet la résolution du problème d'inter-blocage de communication dans un réseau de topologie arbitraire. Grâce à cet outil, nous avons pu concevoir une solution décentralisée et auto-stabilisante au problème de circulation de jeton.

Cette solution a été adaptée pour résoudre le problème d'allocation de ressources dans les réseaux ad-hoc sous une contrainte de mobilité souple. Nous avons à travers des expérimentations fourni une mesure des performances de notre algorithme et plus généralement des solutions fondées sur des marches aléatoires.

Comme nous l'avons précisé dans l'état de l'art, dans les réseaux dynamiques et plus particulièrement dans les réseaux ad-hoc, il n'y a pas de solutions idéales, tout n'est qu'une question de compromis entre performances et gestion de la mobilité. Certains algorithmes se basent sur la construction et la maintenance d'une structure virtuelle sur les différents nœuds, pour parvenir à l'élaboration de solutions très performantes en termes de temps de parcours. L'approche que nous avons proposée, c'est-à-dire l'utilisa-

tion de marches aléatoires, ne fournit pas des performances optimales, les complexités en moyenne pour des parcours ou pour la visite de nœuds sont en $O(n \log n)$. Des algorithmes déterministes résolvent ces problèmes en $O(n)$. Néanmoins, ces solutions déterministes doivent proposer un protocole sous-jacent qui permet la gestion de la mobilité, ce qui alourdit considérablement le système. Notre solution se décharge de cette gestion : il n'y a pas de surcoût supplémentaire excepté lorsque nous proposons en outre la gestion des défaillances transitoires. Nous avons simulé notre algorithme d'allocation de ressources dans le modèle de mobilité le plus général : le modèle de mobilité basé sur une marche aléatoire bornée dans une zone de simulation. Les performances qu'offre notre solution restent du même ordre de grandeur que les performances d'algorithmes basés sur une structure virtuelle.

Nous envisageons maintenant l'écriture d'algorithmes dans un modèle plus proche des réseaux ad-hoc et en particulier de réseaux de capteurs. En effet, lors de l'émission d'un message, dans notre modèle, seul un périphérique reçoit celui-ci, or ces réseaux fonctionnent avec des ondes radio et donc tous les voisins de ce périphérique reçoivent ce message. Il ne s'agit pas non plus d'effectuer une inondation totale du réseau, mais plus raisonnablement d'utiliser cette diffusion locale pour éventuellement transmettre une information. Nous pensons en particulier à l'élaboration d'un algorithme d'élection et de nommage dans un réseau ad-hoc anonyme. Nous réaliserons aussi des simulations pour analyser la performance de cette hybridation entre marche aléatoire et "inondation locale".

La taille des réseaux devient maintenant un facteur limitant pour les algorithmes distribués, notre paradigme de conception d'algorithmes distribués pour les réseaux dynamiques n'échappe pas à cette limitation. Comme nous l'avons vu grâce à notre méthode automatique de calcul de complexité, la plupart des grandeurs caractéristiques d'une marche aléatoire admettent une borne de l'ordre de $O(n^3)$. Cela reste acceptable pour des réseaux de taille moyenne, mais trop important dans un large réseau dès lors qu'il est question de qualité de service. Pour la réalisation de certaines tâches, l'utilisation de multi-marches semble convenir : nous avons évalué par simulation le temps qu'une information distribuée géographiquement soit diffusée à l'ensemble des composants du système, celui-ci semble être inversement proportionnel au nombre de jetons. Pour la réalisation d'autres types de tâches, cette stratégie peut s'avérer inadaptée. Il convient alors pour conserver les performances acceptables de nos algorithmes, de fractionner de manière adaptative ces réseaux de grande taille et d'adapter nos algorithmes en les hiérarchisant sur ces partitions de réseaux. En particulier, notre algorithme de maintenance de structure pourrait être adapté : dans chaque partition circulerait un jeton chargé de construire et de maintenir une structure couvrante sur la partition qui le contient. A un niveau supérieur, un autre jeton circulerait entre les différentes partitions, afin d'échanger les informations entre celles-ci. Nous aurions donc une structure distribuée hiérarchiquement au sein de plusieurs jetons.

Bibliographie

[AB93] Y Afek and GM Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7(1) :27–34, 1993.

[AG94] A Arora and MG Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9) :1026–1038, 1994.

[AH93] E Anagnostou and V Hadzilacos. Tolerating transient and permanent failures. In *WDAG93 Distributed Algorithms 7th International Workshop Proceedings, Springer LNCS :725*, pages 174–188, 1993.

[AKL+79] R. Aleliunas, R. Karp, R. Lipton, L. Lovasz, and C. Rackoff. Random walks, universal traversal sequences and the complexity of maze problems. In *20th Annual Symposium on Foundations of Computer Science*, pages 218–223, 1979.

[And98] Artur Andrzejak. *Lectures on Proof Verification and Approximation Algorithms*, volume 1367 of *LNCS*, chapter 2 : Introduction to Randomized Algorithms, page 29. Springer, 1998.

[APSVD94] B Awerbuch, B Patt-Shamir, G Varghese, and S Dolev. Self-stabilizing by local checking and global reset. In *WDAG94 Distributed Algorithms 8th International Workshop Proceedings, Springer LNCS :857*, pages 326–339, 1994.

[BBBS03] T Bernard, A Bui, M Bui, and D Sohier. A new method to automatically compute processing times for random walks based distributed algorithm. In M. Paprzycki, editor, *ISPDC 03, IEEE International Symposium on Parallel and Distributed Computing Proceeding*, volume 2069, pages 31–36. IEEE Computer society Press, 2003.

[BBF04a] T Bernard, A Bui, and O Flauzac. Topological adaptability for the distributed token circulation paradigm in faulty environment. In Jiannong Cao, Laurence T. Yang, Minyi Guo, and Francis Lau, editors, *ISPA'04, Internationnal Symposium on Parallel and Distributed Processing and Applications, Hong-Kong, China*, volume 3358, pages 146–155. Springer Verlag, December 2004.

[BBF04b] T Bernard, A Bui, and O Flauzac. Random distributed self-stabilizing structures maintenance. In F.F. Ramos, H. Unger, and V. Larios, editors, *ISADS'04, IEEE Internationnal Symposium on Advanced Distributed Systems Proccedings*, volume 3061, pages 231–240. Springer Verlag, january 2004.

[BBFNar] T. Bernard, A. Bui, O. Flauzac, and F. Nolot. A multiple random walks based self-stabilizing k-exclusion algorithm in ad-hoc networks. In *ISADS06 Fifth IEEE Internationnal Symposium on Advanced Distributed Systems*, 2006 to appear.

[BBFR06a] T. Bernard, A. Bui, F. Flauzac, and C. Rabat. Gestion de la mobilité dans un réseau orienté à l'aide d'un mot circulant. In *Actes de la ROADEF, groupe Système Complexe et Décision Distribuée*, 2006.

[BBFR06b] T. Bernard, A. Bui, O. Flauzac, and C. Rabat. Decentralized resources management for grid. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *RDDS'06 International Workshop on Reliability in Decentralized Distributed systems*, volume LNCS 4278, pages 1530–1539. Springer, 2006.

[BBSB04] M. Bui, T. Bernard, D. Sohier, and A. Bui. Random walks in distributed computing, a survey. In Thomas Böhme, Victor M. Larios Rosillo, Helena Unger, and Herwig Unger, editors, *I2CS, 4th International Workshop Innovative Internet Community Systems*, volume 3476, pages 1–14, 2004.

[BCV04] L. Blin, A. Cournier, and V. Villain. An improved snap-stabilizing pif algorithm. In *SSS'04 Self-Stabilizing Systems*, volume 2704 of *LNCS*, pages 199–214. Springer, 2004.

[BDDN01] Marc Bui, Sajal K. Das, Ajoy Kumar Datta, and Dai Tho Nguyen. Randomized mobile agent based routing in wireless networks. *International Journal of Foundations of Computer Science*, 12(3) :365–384, 2001.

[BDF01] Azzedine Boukerche, Sajal K. Das, and Alessandro Fabbri. Analysis of a randomized congestion control scheme with dsdv routing in ad hoc wireless networks. *J. Parallel Distrib. Comput.*, 61(7) :967–995, 2001.

[BDPV99] A Bui, AK Datta, F Petit, and V Villain. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the Fourth Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems)*, pages 78–85. IEEE Computer Society, 1999.

[BDV05] D. Bein, A.K. Datta, and V. Villain. Snap-stabilizing optimal binary search tree. In *SSS'05 Self-Stabilizing Systems*, volume 3764 of *LNCS*, pages 1–17. Springer, 2005.

[BGJDL02] J Beauquier, M Gradinariu, C Johnen, and J Durand-Lose. Token-based self-stabilization uniform algorithms. *Journal of Parallel and Distributed Computing*, 62(5) :899–921, 2002.

[BGK98] J Beauquier, C Genolini, and S Kutten. k-stabilization of reactive tasks. In *PODC98 Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, page 318, 1998.

[BHWG99] Nader H. Bshouty, Lisa Higham, and Jolanta Warpechowska-Gruca. Meeting times of random walks on graphs. *Information Processing Letters*, 69(5) :259–265, 1999.

[BIZ89] J Bar-Ilan and D Zernik. Random leaders and random spanning trees. In *WDAG89*, pages 1,12, 1989.

[BKP03] C. Basile, M.O. Killijian, and D. Powell. A survey of dependability issues in mobile wireless networks. Technical report, LAAS, 2003.

[BP89] JE Burns and J Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11(2) :330–344, 1989.

[Bro89] AZ Broder. Generating random spanning trees. In IEEE Computer Society, editor, *FOCS 89 Proceedings of the 29st annual IEEE Symposium on foundation of computer sciences*, pages 442–447, 1989.

[BS04] Alain Bui and Devan Sohier. Hitting times computation for theoretically studying peer-to-peer distributed systems. In *IPDPS'04 18th International Parallel and Distributed Processing Symposium*, page 179, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[BS05] A. Bui and D. Sohier. On time analysis of random walk based token circulation algorithms. In *ISSADS'05 International School and Symposium on Advanced Distributed Systems*, volume LNCS 3563, pages 63–71. Springer, 2005.

[BV95] S. Bulgannawar and N.H. Vaidya. A distributed k-mutual exclusion algorithm. In IEEE Computer Society Press, editor, *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, page 153, 1995.

[CBD02] T. Camp, J. Boleng, and V. Davies. A survey of mobility models for ad hoc network research. *Wireless Communications and Mobile Computing (WCMC) : Special issue on Mobile Ad Hoc Networking : Research, Trends and Applications*, 2(5) :483–502, 2002.

[CDCD05] W. Choi, S. Das, J Cao, and A. Datta. Randomized dynamic route maintenance for adaptative routing multihop mobile ad hoc networks. *Journal of Parallel and Distributed Computing*, 65 :107–123, 2005.

[CDV05] A. Cournier, S. Devismes, and V. Villain. A snap-stabilizing dfs with a lower space requirement. In *SSS'05 Self-Stabilizing Systems*, volume 3764 of *LNCS*, pages 33–47. Springer, 2005.

[CFS96] Don Coppersmith, Uriel Feige, and James Shearer. Random walks on regular and irregular graphs. *SIAM Journal on Discrete Mathematics*, 9(2) :301–308, 1996.

[CR79] E.J.H. Chang and R. Roberts. An improved algorithm for decentralized extrema finding in circular arrangement process. *Communications of the Association of the Computing Machinery*, 22 :281–683, 1979.

[CR83] O.S.F. Carvalho and G. Roucairol. On mutual exclusion in computer networks. *Communications of the ACM*, 26(2) :145–147, 1983.

[CRR+97] Ashok K. Chandra, Prabhakar Raghavan, Walter L. Ruzzo, Roman Smolensky, and Prasoon Tiwari. The electrical resistance of a graph captures its commute and cover times. *Computational Complexity*, 6(4), 1997.

[CT91] TD Chandra and S Toueg. Unreliable failure detectors for asynchronous systems. In *PODC91 Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 325–340, 1991.

[CTW93] D Coppersmith, P Tetali, and P Winkler. Collisions among random walks on a graph. *SIAM Journal on Discrete Mathematics*, 6(3) :363–374, 1993.

[CW02] Yu Chen and Jennifer L. Welch. Self-stabilizing mutual exclusion using tokens in mobile ad hoc networks. In *Proceedings of the 6th international workshop on Discrete algorithms and methods for mobile computing and communications*, pages 34–42. ACM Press, 2002.

[CW05] Y. Chen and J.L. Welch. Self stabilizing dynamic mutual exclusion for mobile ad hoc network. *Journal of Parallel and Distributed Computing*, 65 :1072–1089, 2005.

[Dav00] V. Davies. Evaluating mobility models within an ad hoc network. Master's thesis, Colorado School of Mines, 2000.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the Association of the Computing Machinery*, 1987.

[DFG83] E.W. Dijkstra, W.H.J. Feigen, and A.J.M. Van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5) :217–219, 1983.

[Dij74] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17(11) :643–644, 1974.

[DJPV00] Ajoy K. Datta, Colette Johnen, Franck Petit, and Vincent Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. *Distributed Compututing*, 13(4) :207–218, 2000.

[DKR82] D. Dolev, M. Klawe, and M. Rodeth. An $o(n \log n)$ unidirectionnal distributed algorithm for the extrema-finding in a circle. *Journal of Algorithms*, 3 :245–260, 1982.

[Dol00] S Dolev. *Self-Stabilization*. MIT Press, 2000.

[DS80] E.W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1) :1–4, 1980.

[DS00] Peter G. Doyle and J. Laurie Snell. *Random Walks and Electric Networks*. 2000.

[DSW02] Shlomi Dolev, Elad Schiller, and Jennifer L. Welch. Random walk for self-stabilizing group communication in ad-hoc networks. In *Proc. 21st Symposium on Reliable Distributed Systems*, 2002.

[Fei95a] U Feige. A tight lower bound for the cover time of random walks on graphs. *Random structures and algorithms*, 6(4) :433–438, 1995.

[Fei95b] U Feige. A tight upper bound for the cover time of random walks on graphs. *Random structures and algorithms*, 6(1) :51–54, 1995.

[FK99] I. Foster and C. Kesselman, editors. *The Grid : Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann, September 1999.

[Fla00] O Flauzac. *Conception d'algorithmes distribués de routage tolèrants aux fautes*. PhD thesis, Université Technologique de Compiègne, 2000.

[Fla01] O Flauzac. Random circulating word information management for tree construction and shortest path routing tables computation. In *On Principle Of DIstributed Systems*, pages 17–32. Studia Informatica Universalis, 2001.

[FLBB89] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Distributed fifo allocation of identical ressources using small shared space. *Transaction on Programmning Languages and systems*, 11(1) :90–114, janvier 1989.

[FLP85] M. Fischer, N. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association of the Computing Machinery*, 32 :374–382, 1985.

[FV97] O Flauzac and V Villain. An implementable dynamic automatic self-stabilizing protocol. In *I-SPAN'97, Third International Symposium on Parallel Architectures, Algorithms and Networks Proceedings, IEEE Computer Society Press*, pages 91–97. IEEE Computer Society Press, 1997.

[GAGPK01] Tom Goff, Nael B. Abu-Ghazaleh, Dhananjay S. Phatak, and Ridvan Kahvecioglu. Preemptive routing in ad hoc networks. In *MobiCom '01 : Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 43–52, New York, NY, USA, 2001. ACM Press.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[GM91] MG Gouda and N Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4) :448–458, 1991.

[Gou95] M. Gouda. The triumph and tribulation in system stabilization. In *WDAG95, 9th Internationnal Workshop on Distributed Algorithm*, volume LNCS :972, pages 1–18. Springer Verlag, 1995.

[HGPC99] Xiaoyan Hong, Mario Gerla, Guangyu Pei, and Ching-Chuan Chiang. A group mobility model for ad hoc wireless networks. In *MSWiM '99 : Proceedings of the 2nd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 53–60, New York, NY, USA, 1999. ACM Press.

[HNM99] RR Howell, M Nesterenko, and M Mizuno. Finite-state self-stabilizing protocols in message-passing systems. In *Proceedings of the Fourth Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems)*, pages 62–69. IEEE Computer Society, 1999.

[HV01] R. Hadid and V. Villain. A new efficient tool for the design of self-stabilizing l-exclusion algorithms : the controller. In *WSS'01, LNCS 2194*. Sringer, 2001.

[HZ01] Hossam Hassanein and Audrey Zhou. Routing with load balancing in wireless ad hoc networks. In *MSWIM '01 : Proceedings of the 4th ACM in-*

ternational workshop on Modeling, analysis and simulation of wireless and mobile systems, pages 89–96, New York, NY, USA, 2001. ACM Press.

[IJ90] Amos Israeli and Marc Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *9th ACM symposium on Principles of distributed computing*, pages 119–131, 1990.

[IKOY02] S. Ikeda, I. Kubo, N. Okumoto, and M. Yamashita. Fair circulation of a token. *Transactions on Parallel and Distributed Systems*, 13(4) :367–372, April 2002.

[IR81] A. Itai and M. Rodeh. Symmetry breaking in distributive networks. In *Proceedings of the Symposium on Theory of Computing*, pages 150–158, 1981.

[JJLV00] J.K., J.H.Kim, L.Lovasz, and V.H.Vu. The cover time, the blanket time, and the matthews bound. In *IEEE Symposium on Foundations of Computer Science*, pages 467–475, 2000.

[JM96a] David B Johnson and David A Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[JM96b] D.B. Johnson and D.A. Maltz. *Mobile computing*, chapter 5 : The dynamic source routing in ad hoc wireless networks. Kluwer Academic Publisher, 1996.

[KKM90] E. Korach, S. Kutten, and S. Moran. A modular technique for the design of efficient leader finding algorithms. *ACM Transaction on Programmning Languages and systems*, 12 :84–101, 1990.

[KP93] S Katz and KJ Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7 :17–26, 1993.

[KR93] AR Karlin and P Raghavan. Random walks and undirected graph connectivity : a survey. In *IMA's, discret probability and algorithm workshop*, 1993.

[KS76] J. G. Kemeny and J. L. Snell. *Finite Markov Chains*. Springer-Verlag, 1976.

[KY97] H Kakugawa and M Yamashita. Uniform and self-stabilizing token rings allowing unfair daemon. *IEEE Transactions on Parallel and Distributed Systems*, 8(2) :154–162, 1997.

[KY04] H. Kakugawa and M. Yamashita. A dynamic reconfiguration tolerant self-stabilizing circulation algorithm in ad-hoc network. In *OPODIS'04, On Principles Of DIstributed Systems*. Springer Verlag, december 2004.

[Lam84] L Lamport. Solved problems, unsolved problems and non-problems in concurrency, invited address. In *PODC84 Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pages 1–11, 1984.

[Lav86] I Lavallée. Contribution à l'algorithmique parallèle et distribuée, application à l'optimisation combinatoire, thèse d'état, 1986. Université de Paris XI, Orsay.

[LeL77] G. LeLann. Towards a formal approach. In B.Gilchrist, editor, *Information Processing '77*, pages 155–160, 1977.

[LH03] Ben Liang and Zygmunt J. Haas. Predictive distance-based mobility management for multidimensional pcs networks. *IEEE/ACM Trans. Netw.*, 11(5) :718–732, 2003.

[Mar85] Alain J. Martin. Distributed mutual exclusion on a ring of processes. *Sci. Comput. Program.*, 5(3) :265–276, 1985.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithm*. Cambridge University Press, 1995.

[MS79] PM Merlin and A Segall. A fail safe distributed routing protocol. *IEEE Transactions on Communications*, 1979.

[MW87] S. Moran and Y. Wolfstahl. Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, pages 145–151, 1987.

[NG98] J. Norris and R. Gill. *Markov Chains*. Cambridge University Press, 1998.

[Per00] C.E. Perkins. *Ad hoc networking*. Addison Wesley, 2000.

[Pet82] G.L. Peterson. An $o(n \log n)$ unidirectionnal algorithm for the circular extrema problem. *ACM Transaction on Programmning Languages and systems*, 4 :758–762, 1982.

[PSM+04] Ravi Prakash, André Schiper, Mansoor Moshin, David Cavin, and Yoav Sasson. A lower bound for broadcasting in mobile ad hoc networks. Technical Report 200437, EPFL, 2004.

[PV97] F Petit and V Villain. A space-efficient and self-stabilizing depth-first token circulation protocol for asynchronous message-passing systems. In *Euro-Par'97 Parallel Processing, Proceedings LNCS :1300*, pages 476–479. Springer-Verlag, 1997.

[RA81] G. Ricart and A.K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1) :9–17, 1981.

[Rab05] C. Rabat. Simulateur dasor : http ://cosy.univ-reims.fr/∼crabat/dasor, 2005.

[Rak94] H.M. Rakotoarisoa. *Simulation distribuée de réseaux de files d'attente.* PhD thesis, Université de Nice Sophia-Antipolis, 1994.

[Ray99] J.W.S. Rayleigh. On theory of resonance. In *Collected scientific papers*, pages 33–75. 1899.

[Ray91a] M. Raynal. *La communication et le temps dans les réseaux et les systèmes répartis*. Eyrolles, 1991.

[Ray91b] Michel Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *SIGOPS Oper. Syst. Rev.*, 25(2) :47–50, 1991.

[RMSL01] E. Royer, P.M. Melliar-Smith, and L.Moser. An analysis of the optimum node density for an ad hoc mobile networks. In IEEE CS, editor, *ICC'O1, IEEE International Conference on Communications*, 2001.

[Sch93] M Schneider. Self-stabilization. *ACM Computing Surveys*, 25 :45–67, 1993.

[Seg83] A. Segall. Distributed network protocols. *Transaction on Information Theory*, 29 :23–35, 1983.

[SM01] Miguel Sanchez and Pietro Manzoni. Anejos : a java based simulator for ad hoc networks. *Future Gener. Comput. Syst.*, 17(5) :573–583, 2001.

[Soh05] Devan Sohier. *Marches aléatoires dans les systèmes complexes : exemples du contrôle aérien et des algorithmes distribués à bases de marches aléatoires.* PhD thesis, Ecole Pratiques des Hautes Etudes, 2005.

[Taj77] WD Tajibnapis. A correctness proof of a topology information maintenance protocol for a distributed computer network. *Communications of the Association of the Computing Machinery*, 20 :477–485, 1977.

[Tar95] G. Tarry. Le problème des labyrinthes. *Nouvelles annales de Mathématique*, 14, 1895.

[Tel94] G Tel. *Introduction to distributed algorithms.* Cambridge University Press, 1994.

[Tet91] P Tetali. Random walks and effective resistance of networks. *J. Theoretical Probability*, 1 :101,109, 1991.

[Tol99] V. Tolety. Load reduction in ad hoc networks using mobile servers. Master's thesis, Colorado School of Mines, 1999.

[Tou80] S Toueg. An all pairs shortest-path distributed algorithm. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, 1980.

[TW91] P. Tetali and P. Winkler. On a random walk problem arising in self-stabilizing token management. In *10th ACM Symposium on Principles Of Distributed Computing*, pages 273–280, 1991.

[TW93] P. Tetali and P. Winkler. Simultaneous reversible markov chains. In T. Szõnyi ed. D. Miklós, V. T. Sós, editor, *Combinatorics : Paul Erdos is Eighty (vol. 1)*, pages 422–452. János Bolyai Mathematical Society, 1993.

[Var94] George Varghese. Self-stabilization by counter flushing. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 244–253. ACM Press, 1994.

[VT95] A.M. Verweij and G. Tel. A monte carlo algorithm for election. In *Proceedings of the 2nd Colloquium on Structural Information and Communication Complexity*, pages pp 77–88. Carleton University Press, 1995.

[WCM01] J. Walter, G. Cao, and M. Mohanty. A k-mutual exclusion algorithm for ad hoc wireless networks. In *POMC01, Proceedings of the first annual Workshop on Principles of Mobile Computing*, 2001.

[WL02] K.H. Wang and B. Li. Group mobility and partition prediction in wireless ad-hoc networks. In IEEE CS, editor, *ICC'02. IEEE International Conference on Communications*, volume 2, pages 1017–1021, 2002.

[WWV01] Jennifer E. Walter, Jennifer L. Welch, and Nitin H. Vaidya. A mutual exclusion algorithm for ad hoc mobile networks. *Wirel. Netw.*, 7(6) :585–600, 2001.

Marches aléatoires et mot circulant, adaptativité et tolérance aux pannes dans les environnements distribués.

Résumé :

Nous proposons dans ces travaux une étude des marches aléatoires dans l'algorithmique distribuée pour les réseaux dynamiques. Nous montrons dans un premier temps que les marches aléatoires sont un outil viable pour la conception d'algorithmes distribués. Ces algorithmes reposent principalement sur les trois propriétés fondamentales des marches aléatoires (Percussion, Couverture, Rencontre). Nous fournissons une méthode qui évalue le temps écoulé avant que ces trois propriétés soient vérifiées. Cela nous permet d'évaluer de la complexité de nos algorithmes. Dans un second temps, nous proposons l'utilisation d'un jeton circulant aléatoirement sous forme de *mot circulant* afin de collecter sur ce jeton des informations topologiques. Ces informations permettent la construction et la maintenance d'une structure couvrante du réseau de communication. Ensuite, nous avons utilisé cette structure pour concevoir un algorithme de circulation de jeton tolérant aux pannes pour les environnements dynamiques. Cet algorithme a la particularité d'être complètement décentralisé. Nous proposons dans un dernier temps d'adapter notre circulation de jeton pour proposer une solution au problème d'allocation de ressources dans les réseaux ad-hoc.

Mots-clés : Algorithmes distribués, Marches aléatoires, Tolérance aux pannes, Autostabilisation, Réseaux dynamiques

---

Random Walks and Circulating Word Adaptivity and Fault Tolerance in Distributed Environment

Abstract :

We propose in this work a study of the random walks in the distributed algorithms for dynamic networks. We first show that random walks are a viable tool for the design of distributed algorithms. These algorithms are based on the three fundamental properties of the random walks (Percussion, Coverrage, Meeting). We provide a method which evaluates elapsed time before these properties are checked. This method enables us to evaluate complexity of our algorithms. In the second time, we propose the use of a token circulating randomly as a emph circulating word in order to collect on this token topological information on a message. This information allows the construction and the maintenance of a covering structure of the communication network. Then, we used this structure to design a fault-tolerant algorithm of token circulation in the dynamic environments. This algorithm is completely decentralized. We propose in a last time to adapt our token circulation to propose a solution to the resource allocation problem in the ad hoc networks.

Keywords : Distributed algorithms, Random Walk, Fault Tolerance, Self-Stabilization, Dynamic networks