N° d'ordre :

THÈSE

présentée à

L'UNIVERSITÉ PARIS VIII – SAINT DENIS

pour obtenir le titre de

DOCTEUR DE L'UNIVERSITÉ PARIS VIII

spécialité :

INFORMATIQUE

par

Franck BUTELLE

Sujet de la thèse :

Contribution à l'algorithmique distribuée de contrôle : arbres couvrants avec et sans contraintes.

soutenue le 3 mars 1994 devant le jury composé de :

MM.	**Patrick**	**GREUSSAY**	Président
	Gérard	**FLORIN**	Rapporteurs
	Laurent	**KOTT**	
	Ivan	**LAVALLÉE**	Directeur
	Christian	**LAVAULT**	Examinateurs
	Nicola	**SANTORO**	
	Marc	**BUI**	

Remerciements

Je suis très reconnaissant envers le Professeur Patrick Greussay de l'Université Paris VIII pour avoir accepté de présider ce jury, qu'il en soit ici remercié.

J'aimerais aussi adresser de chaleureux remerciements au Professeur Gérard Florin du Cnam – Paris, qui a accepté d'être rapporteur de ma thèse et dont les conseils avisés ont largement contribué à l'évolution de celle-ci.

Je remercie également le Professeur Laurent Kott de l'Université de Versailles, appartenant à la direction de l'Inria et qui me fait le très grand honneur de bien vouloir être l'autre rapporteur de cette thèse. A travers lui, j'adresse mes remerciements à toutes les personnes de l'Inria Rocquencourt qui ont permis que cette thèse se réalise dans de bonnes conditions, entre autres grâce à l'octroi d'une bourse de thèse.

Que Nicola Santoro, Professeur aux Universités de Carleton (Canada) et Bari (Italie) trouve ici mes sincères remerciements pour l'intérêt qu'il a porté à cette thèse.

Le Professeur Ivan Lavallée de l'Université Paris VIII n'est pas que le directeur de cette thèse, c'est aussi un ami, qu'il soit ici sincèrement remercié pour son enseignement et ses qualités humaines indéniables.

Autre homme de bien dans ce jury, le Professeur Christian Lavault de l'Insa Rennes, a toute ma sympathie et mon respect ; sa très grande disponibilité et sa gentillesse sont reconnues par tous.

Marc Bui, qu'il me semble connaître depuis toujours, est un ami et un collègue de travail au moral à tout épreuves – puissent mes remerciements lui rendre un peu le soutien qu'il m'a prodigué.

Enfin, en dehors de ce jury, je voudrais remercier le Professeur Catherine Roucairol et l'ensemble de l'équipe qui m'a accueilli à l'Inria : la défunte Action ParaDis où la bonne humeur et la joie de vivre ont su rencontrer la rigueur de la science.

Résumé

Nous présentons dans cette thèse une étude sur des algorithmes distribués asynchrones de contrôle.

Un algorithme de contrôle établit une structure virtuelle sur un réseau de sites communicants. Nous faisons le choix de faire un minimum d'hypothèses sur les connaissances de chaque site. De même, nous évitons autant que possible d'utiliser des mécanismes conduisant à des attentes qui peuvent être pénalisantes comme, par exemple, l'utilisation de synchroniseurs. Ces choix conduisent à privilégier les modes de fonctionnement essentiellement locaux. Nous introduisons toutefois une limite à cette démarche, dans ce travail, nous ne considérons que des algorithmes déterministes.

Dans ces circonstances, un problème essentiel de l'algorithmique distribuée est l'établissement d'une structure de contrôle couvrant la totalité du réseau, dans laquelle chaque site distingue certains de ses voisins de façon spécifique.

Nous avons choisi d'étudier plus particulièrement les algorithmes de construction d'arbre couvrant. Nous montrons comment passer de cette structure au problème de l'élection et à d'autres problèmes classiques de l'algorithmique distribuée.

Les algorithmes de construction d'arbre couvrant étudiés sont de deux types : ceux qui gèrent des phases logiques et ceux qui fonctionnent sans ces phases et qui sont résolument plus imprévisibles dans leur fonctionnement. Nous présentons un algorithme de ce type qui a potentiellement une grande résistance aux pannes car, en plus de son fonctionnement caractéristique, il est associé à une élection non prédéterminée par le réseau contrairement à la grande majorité des algorithmes de la littérature.

Nous étudions d'autres algorithmes de construction d'arbre couvrant avec contraintes, ces dernières apportant une plus grande efficacité à la structure de contrôle établie. En particulier, nous considérons la contrainte de poids total minimum qui caractérise plutôt une recherche économique et celle de diamètre minimum qui concerne l'efficacité à la fois en temps mais aussi évidemment en messages. Cette contrainte n'a jamais été étudiée à notre connaissance en algorithmique distribuée. Nous présentons plusieurs algorithmes distribués suivant la présence, ou non, de pannes.

L'étude pratique de ces algorithmes distribués asynchrones sur de grands réseaux nous a conduit à l'élaboration d'un simulateur-évaluateur. Ce dernier, par rapport à ses très nombreux concurrents, à l'avantage d'être très simple, donc aisément adaptable. Pour rendre la simulation plus réaliste et plus rapide nous l'avons aussi implanté sur des machines parallèles. Le même code source pouvant être exécuté sur une machine séquentielle, sur une machine parallèle à mémoire partagée ou encore sur une machine distribuée.

Introduction

Des ordinateurs qualifiés d'hyper-parallèles sont déjà apparus sur le marché, des réseaux internationaux existent mais, même si comme M. Daniel Hillis «père» de la connection machine, nombreux sont ceux qui pensaient qu'en multipliant le nombre des processeurs on multiplie la puissance de calcul, le développement de logiciels adéquats s'avère être une tâche difficile. Le rêve d'une puissance démesurée se dissout face à des problèmes de taille : comprendre, maîtriser et diriger les communications entre des milliers de processeurs. Même si, dans certaines applications (comme par exemple le traitement d'image), des résultats probants ont été obtenus, le problème reste très difficile pour les autres applications et conduit les chercheurs, du mathématicien au programmeur, à de nouveaux modes de pensée.

En effet, depuis l'enfance, l'apprentissage ainsi que la résolution de problèmes sont habituellement décomposés en petites tâches exécutées en séquence, les unes après les autres. Cette technique ne peut pas, en général, être conservée dans l'utilisation de l'hyper-parallélisme ou parallélisme massif.

Un secteur désormais actif de la recherche s'intéresse de près aux algorithmes distribués. Ces algorithmes sont, par définition, des algorithmes destinés à fonctionner en milieu distribué, c'est à dire sur des machines disposant de processeurs sans mémoire commune, ou encore sur des réseaux de sites communicants. Ces environnements peuvent être caractérisés par un graphe $G = (X, U)$, où X est l'ensemble des sommets représentants les sites et U est l'ensemble des arêtes représentant les liens de communication entre ces sites. Un site étant soit un processeur, soit plusieurs, voire un réseau de communication local entre machines distinctes. La seule contrainte, et qui d'ailleurs s'avère généralement réalisée dans la pratique, est l'existence d'une liaison unique entre le réseau (principal) et le site par arête du graphe des communications. Ce site, s'il représente un réseau local, devra distinguer un unique processeur, parmi ceux composant son réseau, pour les communications avec «le monde extérieur» *i.e.* le réseau principal.

Nous considérons ici des réseaux de sites sans mémoire globale et sans horloge globale. Les sites travaillent en parallèle et de façon asynchrone. Toute action d'un site est conditionnée par la réception d'un ou plusieurs messages (on parle alors d'algorithme « message-driven »).

Actuellement, en plus de l'existence de réseaux locaux autant qu'internationaux, on voit arriver sur le marché des machines parallèles qui sont en fait des machines distribuées c'est à dire formées d'un réseau de quelques dizaines à quelques milliers de processeurs indépendants reliés par un réseau d'interconnexion sans mémoire commune (voir par exemple les machines Thinking Machines CM-5, KSR1, IBM SP1 et d'autres dont on présente, dans le chapitre 2, quelques caractéristiques techniques). Ces machines sont donc une justification supplémentaire, si cela était nécessaire, de l'intérêt de l'algorithmique distribuée.

L'un des problèmes majeurs, avec des réseaux de sites sans mémoire commune, est celui de sa structuration virtuelle. De nombreux travaux ont été réalisés, nous avons voulu apporter notre pierre à l'édifice. Nous avons cherché à établir des algorithmes utilisant les hypothèses les plus faibles possible au sens mathématique du terme. C'est à dire que ces algorithmes sont, en grande partie, indépendants du matériel. Ils sont asynchrones (il n'y a pas nécessité d'existence d'une horloge globale) et ne présupposent pas de connaissance sur la topologie du réseau (à part peut-être sa connexité) ni sur sa taille (sauf exception spécifiée dans le texte). Nous ne formons pas non plus d'hypothèse sur l'homogénéité du réseau : les machines qui le composent peuvent être de capacités et de puissance différentes, seul le protocole de communication doit être le même pour toute machine. De plus, nous ne ferons en général pas de restriction sur le nombre et l'identité des initiateurs des algorithmes distribués considérés. En particulier et sauf exception précisée dans le texte, nous ne ferons pas l'hypothèse d'un initiateur unique.

La justification de ces choix tient en ce que l'augmentation prévisible de la taille des réseaux et des machines distribuées, la combinaison des moyens physiques de connexion (câbles coaxiaux, fibres optiques, ondes Hertziennes...) conduit à des topologies physiques imprévisibles, voire même dynamiques. D'autre part, lorsqu'il y a un très grand nombre de processeurs, la probabilité que l'un d'entre eux (au moins) tombe en panne est forte et des algorithmes, dont le fonctionnement est essentiellement local, sont *a priori* plus résistants aux pannes franches de sites et de lignes.

Par ces choix, nous nous plaçons dans la lignée des travaux de I. Lavallée et de C. Lavault [Laval86, Lavau87].

Suivant les auteurs, la structuration virtuelle d'un réseau consiste avec éventuellement quelques hypothèses supplémentaires telles que la connaissance de la topologie du graphe ou du nombre de sommets, en la construction de structures de contrôle sur la totalité des processus, brisant ainsi la symétrie d'origine.

Avec l'établissement de la structure virtuelle, chaque processus acquiert des connaissances supplémentaires sur le réseau, connaissances liées à la structure de contrôle établie. C'est à dire que chaque site distingue certains de ses voisins comme étant privilégiés et jouant des rôles particuliers.

Les structures généralement utilisées sont l'anneau virtuel ou l'Arbre Couvrant voir [Awer87, GaHS83, HeMR87, KoKM90, LaLa89a, LaRo86].

L'Arbre Couvrant (AC) permet la résolution de plusieurs problèmes en algorithmique distribuée tels l'exclusion mutuelle [TrNa87] ou la synchronisation [Awer85c] ; il est lié aussi aux problèmes de calculs distribués de base tels la recherche d'extremum, l'élection et la terminaison distribuée. Nous rappelons de quelle façon ces problèmes se réduisent à la construction d'AC dans le chapitre 3.

Dans le présent travail, nous présentons une étude à la fois pratique et théorique sur de nouveaux algorithmes de construction d'Arbre Couvrant avec ou sans contraintes. Les algorithmes de construction d'AC sont de deux types : ceux qui gèrent des phases logiques et ceux qui fonctionnent sans ces phases et qui sont résolument plus imprévisibles dans leur fonctionnement. Le fait de ne pas pouvoir prévoir le comportement d'un algorithme rend son analyse difficile, mais, du point de vue de la résistance aux pannes, malveillances ou de l'espionnage, cela peut être un atout.

Dans la littérature de l'algorithmique distribuée, le problème de l'élection est souvent lié au problème de la construction d'un Arbre Couvrant sur le réseau. Usuellement, le processus élu est la racine de l'AC et est d'identité maximum (ou minimum).

Dans notre premier algorithme, en plus de la construction d'un Arbre Couvrant sans phases logiques, nous effectuons une élection d'un site dont on ne peut, *a priori*, prévoir l'identité. C'est à dire qu'elle n'est pas liée à la recherche d'un extremum comme la majorité des algorithmes d'élection. Il s'agit là d'une étape importante dans la recherche d'un algorithme élaborant une structure de contrôle réellement résistante aux pannes. En effet, si l'élu est prévisible, il peut être la cible d'actions extérieures et, si c'est toujours le même, sa charge de fonctionnement est plus importante donc sa probabilité de tomber en panne est alors aussi plus importante. Nous montrerons que son analyse théorique dans le pire des cas peut être trompeuse au vu de ses performances pratiques comparables au meilleur algorithme connu qui utilise la notion de jeton et est par conséquent résolument plus « séquentiel ».

Nous étudions d'autres algorithmes de construction d'Arbre Couvrant avec contraintes, ces dernières apportant une plus grande efficacité à la structure de contrôle établie. En particulier, l'accent sera mis sur les contraintes de poids total minimal, qui caractérise plutôt une recherche économique, et celle de diamètre minimal.

Notre second apport est ainsi un algorithme qui construit un Arbre Couvrant de Diamètre Minimum, problème nouveau en algorithmique distribuée et dont l'intérêt réside dans le gain potentiel que représente cette structure. En effet, la contrainte de diamètre minimal minimise dans cette structure les délais de communication entre sites ainsi que les messages. Nous considérons le diamètre D d'un graphe sous sa forme « valuée », c'est à dire qui est la somme des poids des arêtes du plus long des plus courts chemins. Clairement, si les poids représentent des délais de communications, il existe toujours au moins deux sites du réseau qui nécessitent au moins D unités de temps pour le transfert d'informations d'un site à un autre. Si les poids sont tous identiques à 1 alors il faudra au moins D messages pour ce même transfert. La technique employée ainsi que l'algorithme ont fait l'objet d'une communication à une conférence internationale : [BuBu93b].

Nous décrivons ces algorithmes et certains de leurs concurrents, dont nous avons pu révéler certains défauts, dans le chapitre 3. Dans ce même chapitre, nous expliquerons pourquoi le problème de l'élection est équivalent au problème de la construction d'Arbre Couvrant, ce qui nous amènera à considérer un algorithme de parcours de graphe sous de nouveaux aspects.

Après avoir, dans le chapitre 1, précisé les diverses notations et définitions classiques de la théorie des graphes ainsi que quelques propriétés essentielles, le chapitre 2 illustre quelques notions sur les réseaux ainsi qu'une discussion sur les problèmes de pannes. Certains algorithmes du chapitre 3 permettent, avec quelques modifications, une certaine résistance aux pannes, nous présentons, cas par cas, ces améliorations.

Dans le chapitre 4, nous étudions la simulation d'algorithmes distribués et nous décrivons le simulateur que nous avons développé. Même si de nos jours les machines distribuées commencent à être réellement accessibles, au début de ce travail nous n'avions pas accès à une machine disposant d'assez de processeurs indépendants pour supporter des exécutions intéressantes (un réseau de plus de 400 sites) de nos algorithmes. Ce simulateur garde un intérêt particulier : celui de la mise au point des programmes, celle-ci est en effet grandement facilitée par un tel outil (pas de rediffusion de code ou de compilations multiples sur des sites hétérogènes...). De plus, il nous a apporté une expérience pratique dans la programmation de machines parallèles diverses.

Ce simulateur a en effet été écrit de façon modulaire, ainsi le même source en langage C produit des exécutables qui fonctionnent en parallèle asynchrone sur une Sequent Balance 8000 (disposant de 10 processeurs), sur une KSR1 (disposant de 32 processeurs) mais aussi en séquentiel sur une station SUN. Le choix du parallélisme a amélioré le réalisme de la simulation et la vitesse d'exécution.

Chapitre 1

Généralités

1.1 Introduction

Nous présentons dans ce chapitre quelques définitions et notations bien connues de la théorie des graphes ainsi que le modèle de système distribué asynchrone.

Les algorithmes distribués étant destinés à fonctionner sur des réseaux, ces réseaux sont associés à des graphes, les liens de communications étant les arêtes et les sites les nœuds du graphe. Nous allons reprendre des propriétés essentielles de la théorie des graphes et préciser celles qui nous intéressent. Nous discutons en particulier de la notion de distance et de ses dérivés.

Ensuite, nous présentons le modèle général de système distribué asynchrone, modèle repris par de nombreux auteurs avec certaines nuances. Nous précisons quelles sont les hypothèses précises qui forment le cadre de notre travail. Ces notions essentielles ajoutées à la terminaison et à la mesure de la complexité des algorithmes distribués sont évoqués et discutés.

1.2 Définitions et notations de la théorie des graphes

Nous présentons dans cette section les notations usuelles de la théorie des graphes (voir par exemple C. Berge [Berg70]) plus quelques autres que nous rappelons par la suite avant leur utilisation.

Un *graphe simple* est un graphe dans lequel tout arc relie deux *sommets* distincts, et étant donné deux sommets de G, il existe au plus un arc les reliant. Tous les graphes considérés dans ce document sont des graphes simples.

Il existe deux terminologies distinctes suivant que le graphe est orienté ou non. Nous utilisons les deux terminologies indépendamment car nous ne considérons que des graphes non-orientés et il est évident que ses derniers sont des cas particuliers des graphes orientés et ainsi l'abus de langage qui s'en suit n'est pas très conséquent, du moins pour ce qui nous concerne.

Soit $G = (X, U)$ un graphe *non-orienté* (ou *symétrique*) tel que $|X| = n$ et $|U| = m$. X est l'ensemble des nœuds et U est l'ensemble des arêtes (ou arcs). Nous associons au graphe G, une fonction de coût w (ou de poids, ou encore de délai de communication entre sites pour un réseau) de U vers $\mathbb{R}^+$.

Un graphe est dit *complet* si tout nœud (ou sommet) est relié à tout autre : $\forall x \in X$, $\forall y \in X$, $x \neq y$, $(x, y) \in U$.

En plus des symboles classiques rappelés en annexe A, nous utilisons les notations communes suivantes :

$\boldsymbol{\Gamma_G(x)}$ Ensemble des *voisins* de x.

$\boldsymbol{\delta(x)}$ *Degré* d'un sommet x, égal ici à $|\Gamma_G(x)|$.

$\boldsymbol{[x_1, x_2, \ldots, x_p]}$ = $(u1, \ldots, u_{p-1})$ *Chemin* contenant les sommets x_1 à x_p, les arêtes $u_i = (x_i, x_{i+1})$, $(i \in \{1, \ldots p-1\})$ appartiennent toutes à U. Nous ne considérons dans cette étude que des chemins *élémentaires*, c'est à dire ne comportant pas deux fois le même sommet (et donc pas deux fois la même arête)[1]. Un *cycle* (ou *circuit*) est un chemin dont les extrémités coïncident.

1.2.1 Notions de longueurs, distances, etc.

Nous étendons la définition de la fonction de «poids» w aux ensembles E d'arêtes : $w(E)$ étant alors la somme des poids des arêtes composant cet ensemble, ainsi la *longueur* d'un chemin $\mu = [x_1, x_2, \ldots, x_p]$ est notée $w(\mu)$. Ceci introduit la notion de «longueur valuée» et influence directement les définitions de distance, rayon, diamètre etc, qui vont suivre. Il est à noter que tous les auteurs n'utilisent

[1] Dans la terminologie des graphes non-orientés, il est en fait plus correct de parler de *chaîne*, mais nous gardons la terminologie chemin, et nous gardons l'idée de direction associée dans la mesure où des messages sont émis d'un site pour arriver à un autre.

pas ces définitions qui se voient ici généralisées aux graphes valués. Nous étendons la définition de longueur sur les graphes non-valués en faisant l'amalgame graphe non-valué, graphe valué uniformément avec des poids égaux à 1. De cette façon, cette définition permet de conserver les définitions usuelles sur les graphes non valués où la distance entre deux sommets est égale au nombre d'arêtes d'un plus court chemin reliant ces sommets.

Cette notion de longueur permet d'introduire des définitions et notations généralisées, c'est à dire définies aussi bien sur des graphes valués que sur des graphes non-valués, comme suit :

$\boldsymbol{d_G(x,y)}$ *Distance* de x à y, égale à la longueur d'un plus court chemin de x à y. Cette longueur est choisie égale à $+\infty$ si un tel chemin n'existe pas et égale à 0 pour la distance d'un nœud à lui même : $d_G(x,x)$.

$\boldsymbol{e_G(v)}$ *Ecartement* (ou *excentricité*, ou *séparation*) d'un sommet v défini comme suit : $e_G(v) = \max_{k\in X} d_G(v,k)$. Tant que les graphes considérés sont connexes, l'excentricité est une valeur finie.

$\boldsymbol{D(G)}$ *Diamètre* du graphe G. Il peut être défini par la longueur du plus long chemin ou par la plus grande excentricité :
$D(G) = \max_{x,y\in X} d_G(x,y) = \max_{v\in X} e_G(v)$. Par extension, nous utilisons le terme de *chemin diamètral* de G pour désigner un plus court chemin entre deux nœuds de longueur égale à $D(G)$.

$\boldsymbol{\rho(G)}$ *Rayon* du graphe. Le rayon d'un graphe est égal à l'excentricité minimale soit : $\rho(G) = \min_{v\in X} e_G(v)$.

$\boldsymbol{C(G)}$ Ensemble des centres du graphe. Un *centre* v_0 du graphe est un sommet d'excentricité minimum, son excentricité $e_G(v_0)$, est donc égale au rayon du graphe. Si $\rho(G)$ est fini alors il existe au moins un centre.

1.2.2 Compléments

Un graphe est dit *connexe* si il existe au moins un chemin menant de tout point à tout autre ($\forall x,y \in X,\ \exists\mu = [x,...,y]$). Par la suite nous ne considérons que des graphes connexes sauf peut-être lorsque que nous considérerons les réseaux dynamiques, certains cas de pannes de lignes pouvant être représentés par des graphes non connexes.

Un *arbre* est un graphe $A = (X(A), U(A))$ qui respecte une des propositions équivalentes suivantes (nous supposons ici que $n = |X(A)| \geq 2$ mais en fait un nœud seul est un arbre particulier) :

- A est connexe et sans cycle
- A est sans cycle et admet $n-1$ arêtes
- A est connexe et admet $n-1$ arêtes
- A est sans cycle et en ajoutant une arête, on crée un cycle (et un seul)
- A est connexe et en supprimant n'importe quelle arête, il n'est plus connexe
- Tout couple de sommets de A est relié par un chemin unique.

Soit $APCC(x)$ l'ensemble des *Arbres des Plus Courts Chemins* de racine x. Les arbres de $APCC(x)$ sont donc tels que :

$$\forall x \in X,\, \forall A \in APCC(x),\, \forall i \in X,\, d_A(x,i) = d_G(x,i)$$

Par extension, nous notons $APCC(G)$ l'union des APCC sur G. Nous utilisons aussi le terme d'*arbre couvrant* pour désigner un arbre de G ayant comme sommets tous les sommets de G et comme arêtes un sous-ensemble de U. Ainsi tout arbre de $APCC(G)$ est, de façon évidente, un arbre couvrant. Puisque nous manipulons surtout des arbres couvrants avec ou sans contraintes, il nous est commode de définir D^*_G pour représenter le diamètre d'un *Arbre Couvrant de diamètre Minimum* sur G. Nous verrons plus en détail les propriétés des ACDM par la suite et des algorithmes de construction dans la section 3.4.

Nous utilisons aussi parfois la version orientée de l'arbre couvrant : une *arborescence couvrante* est définie comme un arbre couvrant muni d'une racine x, c'est à dire qu'à partir de x il existe un chemin vers tous les autres nœuds de l'arborescence. Il est à noter que cette notion n'est utilisée que d'un point de vue logique : c'est l'algorithme qui fournit une orientation aux arêtes, pas le réseau.

Un chemin est dit *Hamiltonien* si il passe une fois et une seule par tous les sommets du graphe et, de même, on utilise le terme de circuit Hamiltonien pour un circuit dont l'ensemble des nœuds est égal à X. De façon évidente il existe des graphes, par exemple sur un arbre, sur lesquels il n'est pas possible de trouver un chemin Hamiltonien ou un circuit Hamiltonien. Par conséquent, il est d'usage de définir un chemin (ou un circuit) *pseudo-Hamiltonien* comme étant un chemin (respectivement un circuit) tel qu'il passe au moins une fois par chaque sommet.

Une chaîne est dit *Eulérienne* si elle passe une fois et une seule par toutes les arêtes du graphe. De même, on définit un cycle Eulérien comme étant une chaîne Eulérienne dont les extrémités coïncident.

Dans la suite de ce travail, le G en indice est omis lorsque cela ne provoque pas d'ambiguïté.

1.3 Quelques propriétés utiles

Nous allons rappeler maintenant quelques propriétés de la théorie des graphes qui nous seront utiles dans la suite de cette thèse.

1.3.1 Propriétés des distances

• *Inégalité triangulaire* (ou Eulérienne) :

$$\forall i, j, k \in X,\ d(i,j) + d(j,k) \geq d(i,k)$$

• Un *sous-chemin* d'un plus court chemin est aussi un plus court chemin (Bellman) :

$$\forall \mu = [x_1, \ldots, x_p],\ x_i \in X,\ \forall i \in \{1, \ldots p\},\ w(\mu) = d(x_1, x_p)$$

$$\Rightarrow d(x_1, x_j) = w([x_1, x_j])\, \forall j \in \{1, \ldots, p\}$$

C'est sur l'exploitation de ces deux propriétés que sont basés tous les algorithmes de plus court chemin.

1.3.2 Propriétés sur les rayons et les diamètres

Dans ce paragraphe, nous allons exprimer quelques inégalités sur les diamètres et les rayons.

Lemme 1 $\rho(G) \leq D(G) \leq 2\rho(G)$

Preuve : La preuve est obtenue aisément par contradiction. On peut noter que les bornes supérieure et inférieure sont atteintes, par exemple dans le cas des graphes non valués, pour respectivement le graphe complet et pour un graphe réduit a une simple arête reliant deux nœuds. □

Si T' est un *Arbre Couvrant de Rayon Minimal* sur G, alors

$$\rho(T') = \rho(G) \text{ et } D(T') \leq 2\rho(G)$$

Si T^* est un *Arbre Couvrant de Diamètre Minimal* sur G alors

$$D(G) \leq D(T^*) = D^* \leq D(T')$$

La borne inférieure est atteinte par exemple lorsque G est un arbre et la borne supérieure est atteinte lorsque G est un graphe complet.

Finalement, nous obtenons l'inégalité suivante :

$$\rho(G) \leq D(G) \leq D^* \leq D(T') \leq 2\rho(G) \tag{1.1}$$

Nous pouvons maintenant énoncer quelques propriétés sur les arbres valués.

Par la définition de $T \in APCC(x)$, nous avons :

$$D(T) = \max_{x,y \in X}(d_T(x,y)) \leq \max_{i,j \in X}(d_G(i,x) + d_G(x,j))$$

Lemme 2 *Un centre x de G est aussi un centre de tout arbre $T \in APCC(x)$.*

Preuve : Si x est un centre de G et $T \in APCC(x)$, alors $\rho(T) = e(x) = \rho(G)$. □

Si toutes les arêtes sont de poids 1, nous pouvons reprendre un très vieux résultat de Camille Jordan, daté de 1869, repris par C. Berge dans [Berg70], chapitre 4, théorème n^o 8.

Théorème 1 *Si G est un arbre, et si $D(G)$ est pair, alors G a un centre unique, par lequel passent tous les chemins de longueur maximum ; en outre on a :*

$$\rho(G) = \frac{1}{2}D(G)$$

Si $D(G)$ est impair, alors G admet exactement deux centres, avec une arête qui les relie et par laquelle passent tous les chemins de longueur maximum ; en outre on a :

$$\rho(G) = \frac{1}{2}\left(D(G) + 1\right)$$

Preuve : La preuve donnée par C. Berge est basée sur une démonstration par récurrence sur la taille des arbres. □

Par conséquent, avec nos notations, nous avons $\rho(T) = \left\lfloor \frac{D(T)+1}{2} \right\rfloor$ pour tout arbre T.

Dans le cas plus général où les arêtes sont de poids positif non nul nous obtenons le résultat suivant :

Lemme 3 *Tout arbre T a exactement un ou deux centres ; si il a exactement un centre alors :*

$$D(T) = 2\rho(T)$$

Si il a exactement deux centres :

$$D(T) < 2\rho(T) < D(T) + 2\beta$$

ou β est le poids de l'unique arête joignant les deux centres.

Preuve : Si l'arbre T n'a qu'un centre, alors il existe au moins deux chemins distincts, de ce centre à deux feuilles de l'arbre, de poids identiques et maximum. Donc le rayon de l'arbre est ce poids et le diamètre est obtenu par la plus longue chaîne égale à $2\rho(T)$.

Si l'arbre a deux centres c_1 et c_2. Démontrons d'abord qu'il ne peut y avoir qu'une seule arête entre ces deux centres, puis nous démontrons l'inégalité prévue par le lemme. Supposons donc, par contradiction, qu'il existe un chemin de longueur minimale avec au moins deux arêtes entre c_1 et c_2 et donc un ou deux centres peuvent être trouves dans ce chemin en lieu et place de c_1 et c_2.

Soit β le poids de l'arête joignant c_1 et c_2, α la longueur du plus long chemin de c_1 a une feuille de son sous-arbre ne contenant pas c_2 et, de même, soit α' la longueur du plus long chemin de c_2 a une feuille de son sous-arbre (ne contenant pas c_1). Supposons par exemple que $\alpha \geq \alpha'$.

Nous obtenons alors la relation suivante :

$$D(T) = \alpha + \beta + \alpha' \text{ et } \rho(T) = \alpha + \beta$$

mais

$$\alpha' + \beta > \alpha \ \ (\text{car } \beta > 0)$$

donc

$$\rho(T) < \frac{\alpha + 3\beta + \alpha'}{2} = \frac{D(T)}{2} + \beta$$

et

$$\rho(T) > \frac{\alpha + \beta + \alpha'}{2} = \frac{D(T)}{2}$$

Il ne nous reste plus qu'à montrer qu'un arbre ne peut avoir plus de deux centres, c'est une démonstration par récurrence analogue à celle décrite par C. Berge dans [Berg70] pour démontrer le théorème de C. Jordan où les poids sont de 1 et les graphes sont orientés. □

Lemme 4 *Quand tous les nœuds de G sont des centres, $D(G) = \rho(G)$ et $\min_{T \in APCC(x)} D(T) = D^*$.*

Preuve : Quand tous les nœuds de G sont des centres, cela signifie que tous les $e(u)$ sont égaux, $\forall u \in U$, donc que le minimum égale le maximum et donc que $\rho(G) = D(G)$.

Le centre d'un ACDM T^* est un nœud de G donc ici un centre x et ce centre est un centre d'un arbre $T \in APCC(x)$ et tout chemin diamétral de T passe par x. Soient i et j des nœuds extrémités d'un chemin diamètral, ils sont donc tels que $d_{T^*}(i,x) + d_{T^*}(x,j) = D^*$. Nous obtenons alors

$$d_{T^*}(x,i) + d_{T^*}(x,j) \geq d(i,x) + d(x,j).$$

Donc nous pouvons construire un APCC dont le diamètre est au plus égal à D^*... $\square$

1.4 Le modèle du système distribué

Nous allons décrire maintenant, dans les paragraphes qui suivent, le cadre de notre travail à savoir ce que peut bien représenter un système distribué (ou réparti) et la notion d'algorithme dans ce système. Ces définitions sont très couramment employées et proviennent, entre autres, des articles et ouvrages de C. A. R. Hoare, G. Le Lann, I. Lavallée, C. Lavault et M. Raynal [Hoar87, LaLa86, Laval90, Lavau87, Lela77, Rayn87, Rayn91].

1.4.1 Le modèle

Le modèle standard de système distribué $\mathbb{S}$ considéré dans la suite est une structure logicielle et matérielle répartie sur un réseau *asynchrone point à point de processus séquentiels communicants*. Tout système distribué est composé d'un ensemble de *sites* reliés entre eux par un *système de communication*. Cette structure est modélisée par un graphe connexe, simple et symétrique (voir les définitions du paragraphe 1.2) $G = (X, U)$ où X est l'ensemble fini des sites de $\mathbb{S}$ et U l'ensemble des lignes de communication. Comme pour tous les graphes de cette étude, et sauf avis contraire, $|X| = n$ et $|U| = m$.

Chaque site possède une mémoire locale non partagée de capacité bornée c, et au moins un processeur. Les seules informations échangées entre les sites sont des messages véhiculés par le réseau. Nous admettrons aussi que chaque site possède un numéro d'identification unique le distinguant de tout autre : son *identité*. L'ensemble des identités dans $\mathbb{S}$ est muni d'un ordre total strict et donc les identités sont, dans ce modèle, bien distinctes. Dans certains algorithmes cette hypothèse peut être relâchée : on parle alors de *réseau anonyme* où les processeurs sont sans véritable distinction (on ne peut plus alors parler d'identité, à la rigueur de numéros). Dans ces réseaux de sites indiscernables, les seuls algorithmes disponibles sont des algorithmes probabilistes (voir par exemple [LaLa91, Lavau87]).

Dans un but de généralisation, nous utilisons aussi bien la notion de site que de processeur, processus, ou encore nœud du graphe des canaux de communication. Un site étant soit un processeur, soit plusieurs, voire un réseau de communication local entre machines distinctes. La seule contrainte, et qui d'ailleurs s'avère généralement réalisée dans la pratique, est l'existence d'une liaison unique entre le réseau (principal) et le site, par arête du graphe. Ce site, s'il représente un réseau local, devra distinguer un unique processeur, parmi ceux composant son réseau, pour les communications avec «le monde extérieur» *i.e.* le réseau principal.

Nous devons insister sur le fait que dans un tel système distribué *asynchrone* $\mathbb{S}$, il n'y a pas de mémoire partagée ni d'horloge globale ce qui entraîne l'absence de toute *variable* ou *état global* accessible par les sites à un instant donné. Ce concept d'instant donné est inutilisable et même paradoxal au vu de la définition du modèle asynchrone. Les seuls événements perceptibles par un processus quelconque sont soit produits par le processus lui-même (envoi de message, calcul local) soit des événements générés par d'autres processus (réception de messages). Nous détaillons plus loin les hypothèses liées à l'asynchronisme de $\mathbb{S}$.

Les messages sont traités dans leur ordre d'arrivée. Si plusieurs messages sont reçus simultanément, ils sont traités dans un ordre arbitraire ou bien suivant l'ordre total sur l'identité des émetteurs de ces messages. Dans la suite nous utilisons indifféremment, et tant qu'il n'y a pas d'ambiguïté possible, les termes de *site*, *processus*, *processeurs* et *sommet*, pour désigner les composants de $\mathbb{S}$ et les *liens de communications* sont aussi appelés *lignes*, *arêtes* ou encore *arcs*. Cette synonymie est rendue possible dans la mesure ou ils se confondent dans l'analyse des algorithmes distribués. Usuellement, deux sommets quelconques reliés par une arête sont dits *voisins*.

Nous considérons essentiellement des réseaux de type *point à point*, c'est à dire vérifiant les trois caractéristiques suivantes :

- *Orientation locale* : tout processus est capable de faire la distinction entre ses *portes* d'entrée-sortie. Ceci ne présume en rien de la connaissance des identités des voisins qui sont *a priori* inconnues. Uniquement dans un but de simplification du code des algorithmes présentés, nous associons une porte d'Entrée/Sortie avec l'identité du voisin correspondant. Notons que la connaissance réelle des identités des voisins est une connaissance en quelque sorte globale, elle peut amener une grande diminution du nombre de messages nécessaires à la résolution de tous les problèmes de l'algorithmique distribuée. Notons encore que pour obtenir une telle information il faut que chaque arête soit parcourue deux fois donc conduit à l'échange de $2m$ messages.

- *Communication sans perte de message* : tout message envoyé à un voisin est reçu si ni la ligne, ni le récepteur ne sont fautifs (très précisément exempts de toute faute) durant la transmission du message (voir la section 2.5 pour une discussion sur les fautes en algorithmique distribuée).

  Rappelons que nous ne pouvons pas prévoir le délai de transmission du message, ainsi cette hypothèse ne fait que stipuler qu'au bout d'un temps fini, en l'absence de faute, tout message envoyé est correctement reçu.

- *Taille bornée des messages* : nous supposons que tout message parcourant le réseau G contient au plus t_G bits, où t_G est une constante prédéfinie du réseau. Plus précisément, cette taille est en $O(f(m, id))$ ou en $O(f(n, id))$ (voir $O(1)$ dans certains cas) où n est le nombre de sites et m le nombre de liens. id est la taille, en nombre de bits, de l'identité maximum d'un site de G. A partir de cette hypothèse, nous pouvons déduire le nombre de messages nécessaires pour transmettre b bits : au plus $\lceil \frac{b}{t_G} \rceil$. Il faut mettre en correspondance cette contrainte et les contraintes réelles que l'on rencontre par exemple dans les réseaux à commutation de paquets.

1.4.2 L'algorithmique distribuée

Un *algorithme distribué* $\mathbb{A}$ sur un système distribué $\mathbb{S}$ est la suite des *transitions locales* à effectuer séquentiellement sur chaque processus p de $\mathbb{S}$ suivant la réception de tel ou tel message à partir d'un état déterminé de p. $\mathbb{A}$ peut être considéré comme une collection de processus qui, par échange d'informations, établissent un calcul ou plus généralement collaborent vers un but commun tout en conservant leur autonomie et leur indépendance. On peut alors définir précisément chaque processus par une *suite d'événements*, un ensemble fini d'états et un ensemble fini de *transitions atomiques* entre événements. Nous schématisons les transitions atomiques de la façon suivante :

$$\langle Q_i,\, E_j \rangle \longrightarrow \langle Q_{i+1},\, E_{j+1} \rangle$$

où Q_i est le i^{e} état du processus et E_j une j^{e} *action atomique* du processus. Nous classons les événements en trois catégories : les événement d'envoi, les événement de réception et les événements internes. Un *événement interne* ne produit qu'un changement d'état, un *événement d'envoi* provoque l'envoi d'un message asynchrone et un *événement de réception* donne lieu à la réception d'un message positionné alors dans un tampon. Si le tampon était vide alors cet événement est lié à un événement interne : la mise à jour de l'état local suivant le contenu du message.

Le concept d'algorithme distribué est donc clairement distinct de celui d'algorithme séquentiel et n'en est pas non plus une variante. Il est clair que le premier recouvre une réalité particulière : il nécessite un système distribué avec ses lignes de communication et ses sites avec leurs mémoires locales et leurs états locaux. Il repose par conséquent essentiellement sur des notions spécifiques aux environnements distribués tels la communication, la causalité entre événements, la connaissance acquise par apprentissage etc.

1.4.2.a Notions de symétrie

Les algorithmes distribués peuvent être plus ou moins répartis, c'est à dire que suivant le degré de symétrie entre les processus nous pouvons établir un classement entre eux. Quatre niveaux de symétrie syntaxique du code se dégagent de la littérature en algorithmique distribuée : la symétrie totale, la symétrie forte, la symétrie de texte, et la non-symétrie (voir J. E. Burns [Burn81]).

- La symétrie totale suppose que tout processus a le même contexte (ensemble de variables) ainsi que le même algorithme séquentiel, cette symétrie sous-entend des comportements parfaitement identiques. Avec de telles hypothèses aucune distinction n'est possible entre sites et donc aucune élection (dans la mesure ou l'on considère toujours des algorithmes déterministes).
- Dans la symétrie forte, les processus peuvent avoir des comportements différents suivant les messages qu'ils reçoivent mais en tout état de cause, le nom du site (son identité) ne doit pas apparaître comme constante du programme. Des algorithmes fonctionnant sur des réseaux anonymes se rangent dans cette classe même si la plupart d'entre eux commencent par tirer au hasard une identité.
- La symétrie de texte illustre le fait que le texte de l'algorithme est le même pour chaque processus mais les identités des processus sont toutes distinctes et cette identité est utilisée au sein de l'algorithme ; ainsi ils peuvent avoir des comportements différents suivant leurs identités et les messages qu'ils reçoivent. Les algorithmes que nous présentons dans le chapitre 3 appartiennent tous à cette classe.
- Enfin, il y a non-symétrie, lorsque chaque processus exécute un algorithme différent (protocoles clients-serveurs etc.).

Ces critères sont, rappelons le, purement syntaxiques, les deux premiers sont d'ailleurs très contraignants. Ainsi, sur des réseaux asynchrones dont la topologie du graphe des communications est inconnue par les sites qui le composent, il n'est pas possible de concevoir d'algorithme distribué déterministe avec une symétrie forte.

Une approche *a contrario* sémantique de la symétrie dans les algorithmes distribués a été étudiée notamment par L. Bougé [Boug87]. Cette approche porte sur la symétrie des rôles que jouent les processus dans le réseau. De fait, la topologie du réseau et la place d'un processus dans ce réseau joue alors un rôle primordial ainsi que la trace de son exécution. La première difficulté est qu'alors, il ne suffit pas de considérer le nombre de voisins d'un processus mais des propriété globales telle que son excentricité, le nombre de voisins de ses voisins etc. Les contraintes sur les position des processus peuvent être exprimés par des automorphismes de graphes, mais en ce qui concerne les traces, la sémantique établie impose de considérer l'espace de tous les calculs, ce qui est en général impossible ; c'est pourquoi nous nous sommes contentés d'utiliser la définition syntaxique.

Les trois premiers modes de programmation peuvent d'ailleurs être regroupés sous le sigle *SPMD* Single Program Multiple Data. Sous ce sigle, sont rangés de très nombreux concepts pas tous compatibles ; nous restreignons ici son emploi aux notions que nous avons présenté plus haut.

1.4.3 Démarrage distribué

Le système $\mathbb{S}$ se comporte selon les règles suivantes :

- Un processus est toujours dans l'un des deux états fondamentaux : *actif* ou *passif*.
- Seuls des processus actifs peuvent émettre des messages.
- Un processus décide de passer de l'état actif à l'état passif suite à un événement interne.
- Un processus passe de l'état passif à l'état actif uniquement sur réception de message ou suite à un événement extérieur.

La dernière règle exprime la notion de démarrage : plus précisément, dans le modèle général, nous ne formons pas d'hypothèse particulière sur l'identité du ou des sites qui démarrent l'algorithme. De même nous ne fixons aucune connaissance sur le nombre de ces sites que l'on qualifie *d'initiateurs*. L'unique exigence est qu'au moins un démarre ; c'est à dire passe de l'état passif à l'état actif.

Il n'existe que deux conditions d'éveils d'un site, la réception d'un message ou un stimulus extérieur. Ce dernier représente une intervention de type mise en circuit ou ordre d'exécution que ce soit d'un utilisateur ou bien d'un mécanisme extérieur quelconque.

Nous devons toutefois accepter une autre hypothèse que l'on retrouve sous-entendue dans la quasi-totalité des algorithmes distribués connus sans pour autant qu'elle soit précisée systématiquement : le démarrage d'un site consiste à l'exécution de la première instruction de l'algorithme distribué. Cette hypothèse semble fort naturelle mais elle peut être discutable surtout lorsque l'on considère les problèmes de reprise sur panne. On peut consulter à ce sujet l'article de M. Fischer *et al.* [FMRT90].

Avec l'hypothèse précédente, il est clair que l'on pourra distinguer éveil spontané et éveil par réception de message par un simple test sur le tampon de réception.

1.4.4 Terminaison distribuée

Avec les règles précédentes, la *terminaison* d'un algorithme distribué peut survenir suivant deux schémas distincts exprimant tous deux la notion de stabilité globale. Le premier respecte le sens usuel : l'algorithme termine si tous les processus passent à l'état passif et restent passifs et savent que tous les autres finissent au bout d'un délai fini par devenir passifs – on utilise alors le terme de *terminaison par processus*. Cette terminaison est très contraignante et peut être difficile à obtenir.

La *terminaison par message* se caractérise par l'absence de communication – elle n'implique pas que l'information de fin d'exécution de l'algorithme soit connue de tous les processus. Ce dernier schéma de terminaison est celui de certains algorithmes de routage (voir par exemple A. Segall [Sega83]) : tant que le réseau reste stable, l'algorithme de routage ne provoque pas d'échange de messages de mise à jour, c'est à dire que l'état global du système est stable et reste stable – autrement dit il s'agit d'une terminaison par défaut de messages.

Dans une terminaison par processus, il est nécessaire qu'un site qui a terminé l'exécution de l'algorithme ne reçoive plus de messages ayant un rapport avec cet algorithme. Dans le cas contraire, il pourrait alors être réveillé par ce message et ne pourrait déterminer si le message fait partie de cet algorithme ou d'une nouvelle exécution.

De nombreux travaux ont été réalisés autour du problème de la détection de la terminaison d'algorithmes distribués, comme le montre F. Mattern dans [Matt87] et la thèse de G. Tel [Tel91].

Nous pouvons maintenant résumer et préciser les hypothèses que nous ferons sur $\mathbb{S}$.

1.4.5 Hypothèses : résumé et précisions

1. Les communications sont asynchrones non bloquantes - Chaque processus dispose d'un tampon borné. (Il est possible d'utiliser un réseau de communication synchrone a condition de pouvoir exécuter plusieurs processus par nœud – voir par exemple la thèse de M. Bui [Bui89]).

2. Les liaisons sont bidirectionnelles (les messages peuvent se croiser sur une ligne).

3. Les délais de transmission sont finis (mais non bornés). Lorsque nous cherchons à estimer la complexité temporelle d'un algorithme, nous fixons artificiellement un délai pour chaque arête.

4. Si des messages arrivent simultanément, ils sont pris en compte séquentiellement.

5. Il n'y a pas de panne de processus.

6. Il n'y a ni perte, ni duplication, ni modification de message.

7. Un sous-ensemble non vide de l'ensemble des processeurs s'éveille spontanément.

8. Il n'y a pas de déséquencement des messages – les messages ne peuvent se doubler sur la ligne ; c'est la discipline PAPS : Premier Arrivé, Premier Servi dite aussi FIFO « First In, First Out » (on peut toujours se ramener à ce cas par l'ajout d'estampilles, voir l'article de L. Lamport [Lamp78]).

9. Les seules connaissances accessibles pour un processus sont son identité et les portes d'Entrée/Sortie (distinctes) qui mènent vers ses voisins (en fait, pour simplifier l'écriture des algorithmes, on ne fait pas dans ce travail de distinction entre la porte qui mène vers un voisin et l'identité du voisin. Cette simplification est possible car les algorithmes que nous présentons ont la particularité d'être insensibles à cette hypothèse).

10. Toutes les identités des processus sont distinctes.

11. Le graphe représentant le réseau des processus est connexe.

Les hypothèses 5 et 6 sont parfois relâchées voire supprimées suivant les algorithmes étudiés ; l'hypothèse 11 peut alors être mise en défaut pendant de courtes durées.

L'hypothèse 1 pose différents problèmes, en fait, la plupart des algorithmes asynchrones ne fixent pas de borne à la taille des tampons. Malgré tout, les algorithmes que nous allons présenter échangent généralement un nombre fini borné de messages, par conséquent une borne sur le nombre de messages échangés est aussi une borne pour la taille des tampons. De fait, les algorithmes échangeant un nombre borné de messages ne nécessitent généralement que des tampons de taille $O(\delta)$ où δ est le degré du réseau.

1.5 Évaluation de la complexité

1.5.1 Notations de Landau

Les notations suivantes sont classiques en théorie de la *complexité*, nous les rappelons ici succinctement. Nous n'étudions que des mesures de coût ou de dénombrement donc *a priori* arithmétiques et positives, c'est à dire de $\mathbb{N}$ dans $]0, +\infty[$.

Soient f et g, deux fonctions arithmétiques à valeurs positives,

$$\begin{array}{rcl} f = o(g) & \iff & \displaystyle\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0 \\ f = O(g) & \iff & (\exists \lambda \in]0, +\infty[)\,(\exists n_0 \in \mathbb{N})\; n > n_0 \Rightarrow f(n) \leq \lambda g(n) \end{array}$$

Dans le cas présent cette deuxième équation s'exprime plus simplement en disant que lorsque n tend vers $+\infty$, il existe un réel λ tel que $f(n) \leq \lambda g(n)$.

$$\begin{array}{rcl} f = \Omega(g) & \iff & (\exists \lambda \in]0, +\infty[)\,(\exists n_0 \in \mathbb{N})\; n > n_0 \Rightarrow f(n) \geq \lambda g(n) \\ & \iff & g = O(f) \\ f = \theta(g) & \iff & (f = O(g)) \wedge (f = \Omega(g)) \end{array}$$

Pour une présentation plus complète de ces notions et de la manipulation des notations asymptotiques voir le livre de C. Froidevaux, M.-C. Gaudel et M. Soria [FrGS90].

1.5.2 Complexité

Pour comparer deux algorithmes séquentiels, l'analyse s'appuie sur le calcul du nombre d'opérations élémentaires nécessaires à l'obtention du résultat en fonction de la taille des données. Les opérations élémentaires sont souvent les opérations arithmétiques, les opérations d'accès à la mémoire etc. En fait les mesures de complexité des algorithmes sont la mesure de quantité de ressources utilisées.

En algorithmique distribuée, le concept est le même et l'on peut résumer comme suit les ressources utilisées par $\mathbb{A}$:

- le temps de calcul local en chaque processus,
- le temps de communication entre processus pour échanger l'information nécessaire à l'exécution des instructions globales de $\mathbb{A}$,
- les messages (ou le nombre de bits) échangés,
- l'information transmise de processus à processus, et l'information disponible sur l'état des processus le long de chaque ligne de communication et en chaque site,
- l'occupation en mémoire.

L'analyse de la complexité va donc quantifier ces ressources pour obtenir des mesures de complexités de $\mathbb{A}$: à la fois en temps de calcul local, en délais de communication, en nombre de messages ou de bits utilisés, en quantité d'information transmise et disponible ainsi que la place mémoire nécessaire.

Dans un environnement distribué asynchrone, l'exécution d'un algorithme est non seulement dépendante de l'entrée comme dans un algorithme séquentiel mais en plus deux exécutions différentes sur les mêmes entrées pourront produire des résultats différents, donc utiliser des quantités de ressources différentes. Cette dernière remarque ajoutée à la caractérisation d'une nouvelle ressource essentielle : la transmission de l'information, explique la difficulté de l'analyse des algorithmes distribués en particulier asynchrones.

La complexité se mesure suivant deux paramètres essentiels : la complexité en *quantité d'information échangée* (voire en nombre de messages dans certains cas) et le temps total d'exécution de l'algorithme. La quantité d'information échangée se calcule en additionnant le nombre de bits de tous les messages échangés. En première analyse elle est donc fortement liée à la complexité en nombre de messages (multiplié par t_G la taille maximum d'un message).

La complexité en temps d'un algorithme distribué $\mathbb{A}$ dans le modèle $\mathbb{S}$ est une mesure paradoxale car le temps lui-même est une notion inutilisable en théorie dans un réseau asynchrone. Toutefois, il est d'usage de définir (uniquement à des fins d'analyse) soit une borne maximum sur le délai d'acheminement d'un message, soit une fonction de cette borne et de la taille du message véhiculé. Il est évident que cela revient à plonger l'algorithme dans un milieu synchrone et ceci peut provoquer des effets pervers sur la mesure de ces algorithmes. En effet, un algorithme synchrone exécuté sur un réseau synchrone est en général plus performant qu'un algorithme asynchrone exécuté sur un réseau asynchrone.

Rappelons ici que le temps d'exécution d'un algorithme asynchrone est en fait non borné puisque les délais de communications sont non bornés. Cela ne signifie pas qu'un algorithme asynchrone met un temps infini à s'exécuter mais simplement que l'on ne dispose pas de données permettant de calculer une borne sur la durée d'exécution.

Nous avons déjà évoqué dans le paragraphe 1.4.2.a un autre critère d'évaluation : le degré de répartition via les quatre niveaux de symétrie. Tous les algorithmes présentés dans ce travail sont de type SPMD, c'est à dire qu'au moins l'hypothèse de symétrie de texte sera toujours vérifiée. Par ailleurs nous formons l'hypothèse que les temps de calcul locaux ainsi que les temps d'attente des messages dans les tampons des sites sont négligeables face aux coûts de communication (délais de transmission et nombre de messages transmis).

Dans un réseau local ou dans une machine parallèle à mémoire partagée virtuelle (voir chapitre 2) les délais de communication sont très faibles (ou tout du moins très peu fluctuants, donc bornés par des fonctions linéaires de la taille des messages véhiculés) par rapport aux temps de calcul local. Notre hypothèse nous place donc au contraire dans le cadre de réseaux non-locaux comportant de nombreux sites géographiquement très dispersés.

1.6 Conclusion du chapitre 1

En fait, un des problèmes les plus exploré de l'informatique distribuée est de briser la symétrie des processus en donnant à l'un d'entre eux un privilège suivant différentes techniques. Ce sont des algorithmes d'élection et d'exclusion mutuelle dans le sens le plus général du terme (lecteurs-rédacteurs, producteur-consommateurs, exclusion mutuelle simple, etc...).

Le problème de l'élection et, plus généralement encore, l'élaboration en commun d'une structure de contrôle couvrant la totalité du réseau jouent un rôle essentiel en algorithmique distribuée. Nous étudions des algorithmes de construction de ces structures dans le chapitre 3.

Nous avons précisé dans ce chapitre le cadre de notre travail en illustrant des notions pour la plupart très connues sur la théorie des graphes et en détaillant nos hypothèses et nos choix. En particulier, nous avons délibérément choisi de ne nous intéresser qu'aux algorithmes distribués asynchrones disposant d'au moins une symétrie de texte. Ce choix est guidé par la réalité des réseaux et machines distribuées comme nous le montrons dans le chapitre suivant. La symétrie de texte permet non seulement une grande souplesse d'utilisation, mais il est inconcevable, pour pouvoir exécuter un seul et même algorithme, de programmer des milliers de processeurs avec des codes distincts !

D'autre part, ces algorithmes se caractérisent par leur difficulté d'appréhension : en effet, le comportement d'algorithmes distribués asynchrones nécessite à la fois une étude théorique longue et difficile, mais aussi de nombreuses exécutions pour véritablement saisir toute la valeur d'un algorithme par rapport à un autre. Une étude seulement théorique où l'on ne figurerait que des résultats austères sans références à l'utilisation possible de ceux-ci serait de peu d'intérêt : nous allons donc examiner dans le chapitre suivant, le support pratique de notre travail : des réseaux désormais classiques et des machines que l'on peut appeler distribuées.

Chapitre 2

Des réseaux

2.1 Introduction

Nous allons présenter dans ce chapitre différentes architectures de réseaux de communications classiques (ou qui le deviennent) puis quelques caractéristiques techniques des nouvelles machines distribuées, utilisant ces architectures.

La troisième partie du chapitre est consacrée, après un rappel succinct de la norme OSI, à la théorie des fautes et des pannes qui peuvent survenir dans les réseaux de communication. Nous présentons alors dans quel cadre nous pouvons réaliser des algorithmes tolérants aux fautes.

2.1.1 Définitions

Un *réseau de communication* est un ensemble de sites pouvant communiquer entre eux. Le réseau est représenté par un graphe $G = (X, U)$, ou tout nœud de X est associé à un *site* et toute arête est associée à un lien de communication (que l'on considére toujours bidirectionnel sauf mention particulière). Dans un but de généralisation, nous utilisons aussi bien la notion de site que de processeur ou de processus, un site étant soit un processeur, soit plusieurs, voire un réseau de communication local entre machines distinctes. La seule contrainte, et qui d'ailleurs s'avère généralement réalisée dans la pratique, est l'existence d'une liaison unique (associée à une arête du graphe) entre le réseau (principal) et le site. Ce site, s'il représente un réseau local, doit distinguer un unique processeur, parmi ceux composant son réseau, pour les communications avec «le monde extérieur» *i.e.* le réseau principal.

Nous avons déjà défini dans le chapitre précédent le *degré* d'un nœud comme le nombre d'arêtes incidentes à ce nœud. Le degré représente le nombre de liens de communication du site, par extension nous définissons le *degré d'un réseau* comme étant le plus grand degré de ses nœuds.

La *distance* entre deux sites est égale à la distance dans le graphe associé, elle a été définie au paragraphe 1.2. Rappelons le, elle est considérée de deux façons différentes suivant que le graphe est valué ou non. Si le graphe est valué, la distance est définie comme la somme des poids des arêtes d'un plus court chemin. Dans le cas contraire, la distance entre deux sommets (ou nœuds) est le nombre de liens du plus court chemin qui les relie. Nous utilisons aussi la *distance moyenne* $\overline{d}$, définie par la moyenne des distances parcourues par un message entre deux sites quelconques. Cette distance dépend du réseau mais aussi de l'algorithme de *routage* (ou de propagation) des messages utilisé. Nous détaillons son calcul par la suite.

Le *diamètre* d'un réseau est le diamètre du graphe associé. C'est la plus grande distance entre deux de ses sommets. Ainsi le diamètre représente un pire des cas et la distance moyenne un cas moyen du délai de communication d'un point à un autre du réseau.

2.1.2 Calcul de la distance moyenne

Pour calculer la distance moyenne il faut définir un routage sur le graphe. Nous allons supposer, dans ce chapitre, que l'algorithme de routage suit simplement les plus courts chemins (en nombre d'arêtes). Ce qui nous permet d'exprimer mathématiquement la distance moyenne en fonction de la distance moyenne $\overline{d_i}$ d'un nœud i vers les autres comme suit :

$$\overline{d} = \frac{1}{n}\sum_{i=1}^{n}\overline{d_i} = \frac{1}{n^2}\sum_{1\leq i,j\leq n} d(i,j)$$

avec

$$\overline{d_i} = \frac{1}{n}\sum_{j=1}^{n} d(i,j)$$

où n est toujours le nombre de nœuds du réseau.

2.2 Quelques réseaux

2.2.1 L'anneau

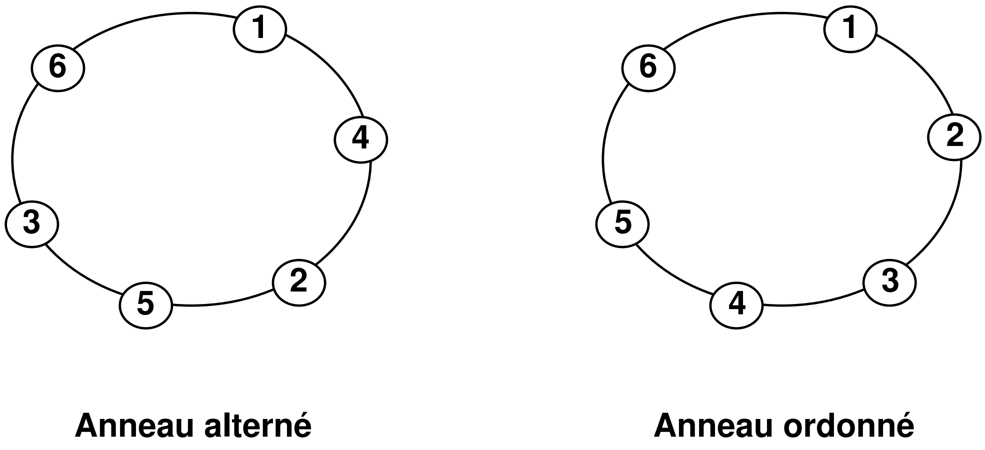

Figure 2.1 : Réseaux en anneaux

Un *anneau* $A(n)$ est un graphe où les n nœuds sont reliés comme dans la figure 2.1.

De façon évidente, le degré de ce graphe est 2, son diamètre et son rayon sont égaux à $\lfloor n/2 \rfloor$. Les distances moyennes $\overline{d_i}$ sont toutes identiques, par conséquent on a : $\overline{d} = \overline{d_1}$. Or,

$$\overline{d_1} = \frac{1}{n} \sum_{j=1}^{n} d(1, j)$$

•Si n est pair, $\overline{d_1} = \frac{1}{n}\left(\frac{n}{2} + 2\sum_{k=1}^{\frac{n}{2}-1} k\right)$, donc $\overline{d} = \frac{n}{4}$.

•Si n est impair, $\overline{d_1} = \frac{2}{n}\sum_{k=1}^{\frac{n-1}{2}} k$, donc $\overline{d} = \frac{n^2-1}{4n}$.

Par conséquent, dans un anneau la distance moyenne varie asymptotiquement comme une fonction linéaire de n ($\overline{d} = \Theta(n)$).

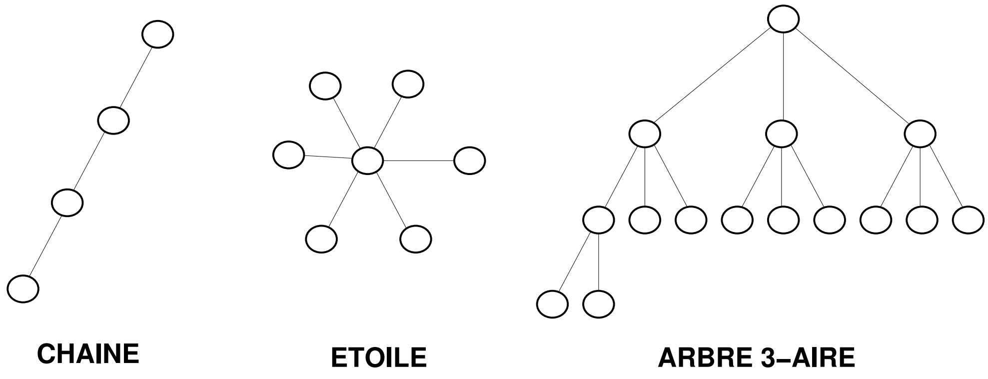

Figure 2.2 : Arbres: chaîne, étoile, équilibré 3-aire

2.2.2 L'arbre et ses variantes

2.2.3 Arbres

Un *arbre T(n)* est un graphe sans cycle où les n nœuds sont reliés comme, par exemple, dans la figure 2.2.

Les informaticiens considèrent plusieurs types d'arbres : un arbre p-aire est un arbre dans lequel chaque noeud a au plus p fils. Par conséquent, l'arbre p-aire a un degré de $p+1$. Le degré des nœuds d'un arbre quelconque peut varier de 1 (chaîne de deux nœuds) à $n-1$ (étoile) ; le diamètre peut alors varier respectivement de n à 1. L'arbre des informaticiens est donc plutôt une arborescence, nous utilisons indifféremment les deux terminologies dans la mesure ou cela ne créé pas d'ambiguïté.

Nous définirons ici la hauteur d'un nœud $h(n)$ par la plus courte distance entre ce nœud et une feuille. Si les hauteurs des fils d'un nœud ne diffèrent pas de plus de 1, ce nœud est dit équilibré. L'*arbre équilibré* est un arbre dans lequel tous les nœuds sont équilibrés. Il peut être *plein* si tous les nœuds non feuilles, sauf peut-être un, ont p fils (voir l'arbre 3-aire de la figure 2.2). Les arbres utilisés dans les réseaux sont généralement pleins, le diamètre d'un arbre p-aire plein est compris entre $2\lfloor \log_p n \rfloor$ et $2\lfloor \log_p n \rfloor + 2$, leur distance moyenne est en $\theta(\log n)$.

Un arbre en étoile est aussi appelé *star graph*. Il est utilisé par exemple pour des réseaux locaux de type maître-esclaves où le site maître est le centre de l'étoile et gère des accès de type entrées-sorties.

2.2.4 Arbre à large bande

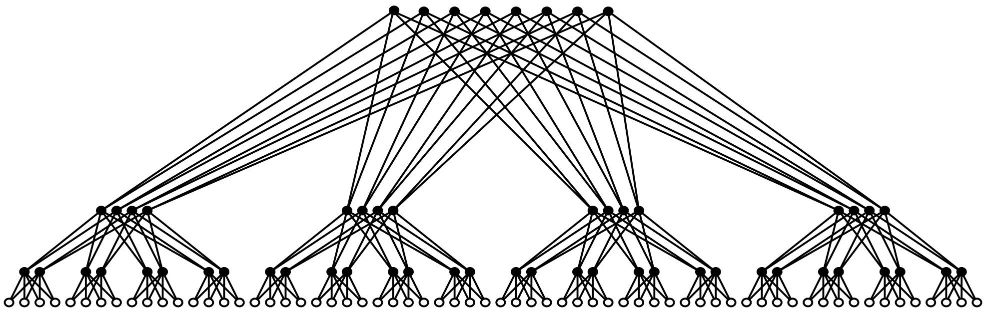

Figure 2.3 : Arbre à large bande d'ordre 64

Cette «variante» de l'arbre a été définie par C. Leiserson dans [Leis85]. Un Arbre à large bande aussi appelé *Fat-Tree* n'a rien à voir, a part peut-être son nom, avec un arbre de la théorie des graphes. Il s'agit de multiples arbres 4-aires superposés de sorte que la section (bande passante) augmente au fur et à mesure que l'on s'approche de la racine. Ainsi, il n'y a pas dégradation des délais de communication avec l'augmentation du nombre de nœuds si le protocole de routage utilisé est correctement réalisé. Les processeurs de calcul se situent au niveau des feuilles de l'arbre (64 dans le cas de la figure 2.3).

Pour aller d'une feuille à une autre il faut parcourir la même distance que dans un arbre 4-aire donc le diamètre est $\lceil \log_4(n) \rceil$ et la distance moyenne est toujours en $\theta(\log n)$.

2.2.5 La grille et ses variantes

2.2.5.a La grille

Une *grille G(p,k)* de dimension p et de base k est un ensemble de $n = k^p$ sites arrangés comme les points de coordonnées entières entre 0 et $k-1$ dans un espace de dimension p (voir le rapport de J.-P. Sansonnet [Sans89] et la figure 2.4) : chaque sommet est connecté avec les voisins dont une coordonnée diffère exactement de 1.

Le degré des nœuds varie de p (dans la périphérie du graphe) à $2p$, ainsi le degré du réseau est-il égal $2p$.

Considérons que les nœuds sont numérotés de 0 à $k^p - 1$, chaque point étant repéré par ses coordonnées dans un espace de dimension p, si a est le numéro d'un

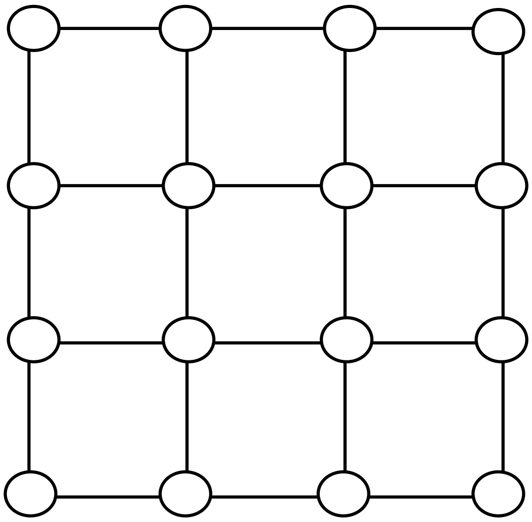

Figure 2.4 : Grille G(2,4)

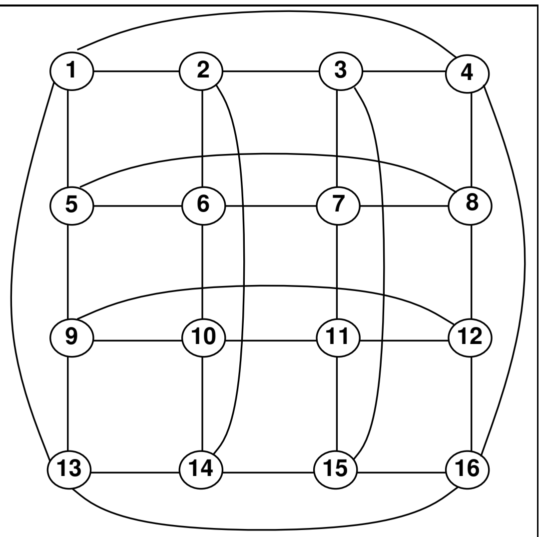

Figure 2.5 : Tore T(2,4)

nœud, a s'écrit soit sous la forme d'un p-uplet $(a_0, a_1, \ldots, a_{p-1})$ soit sous la forme d'un entier : $\sum_{i=0}^{p-1} a_i k^i$ par passage de la base k à la base 10. Ainsi le point de coordonnées $(0, 0, 0, \ldots, 0)$ (p zéros) peut aussi être décrit simplement par l'entier 0 et le point de coordonnées $(1, 1, 1, \ldots, 1)$ (p uns) par l'entier $1 + k + k^2 + \ldots k^p$. Cette double notation est très pratique et très utilisée dans les descriptions de la plupart des topologies ou la notion de dimension intervient.

La distance entre deux sommets est égale à la *distance de Manhattan*[1], qui s'exprime par $d(a, b) = \sum_{i=0}^{p-1} |b_i - a_i|$ (si $b = (b_1, b_2, \ldots, b_p)$).

Le diamètre d'une grille G(p,k) est égal à $p(k-1)$ et est obtenu en considérant, par exemple, la distance entre $(0, \ldots, 0)$ et $(k-1, \ldots, k-1)$ par la distance de Manhattan.

La distance moyenne entre nœuds est de $n(k^2 - 1)/(3k)$ (voir [Sans89]).

2.2.5.b Le tore

Un *tore T(p,k)* de dimension p et de base k est une grille où les processeurs des extrémités (c'est à dire aux coordonnées 0 et $k-1$ de toute dimension) sont reliés

[1] Dans la ville de Manhattan, les rues sont toutes à angle droit, donc pour aller d'un carrefour à un autre dans Manhattan il faut parcourir cette distance, avec $p = 2$.

entre eux [Sans88, Sans89]. Chaque nœud appartient ainsi à p cycles (anneaux) de processeurs, comme le montre la figure 2.5.

Un tore de dimension p a un degré de $2p$. Son diamètre est égal à p fois le diamètre d'un anneau de k nœuds donc $pk/2$ (voir le paragraphe 2.2.1). Pour la même raison, la distance moyenne est égale à p fois la distance moyenne dans un anneau A(k), soit $pk/4$ si k est pair et $p(k^2-1)/(4k)$ si k est impair.

2.2.6 L'hypercube

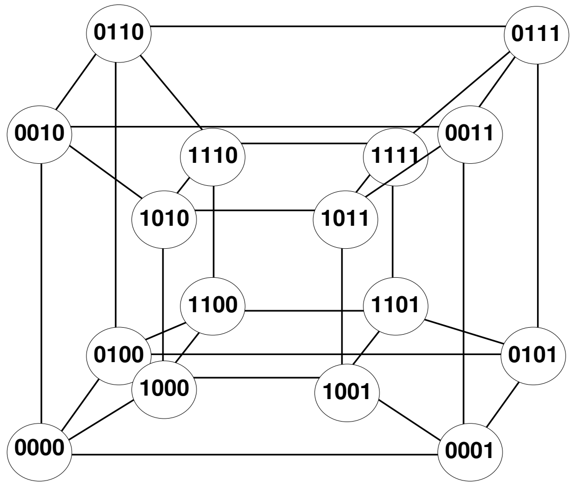

Figure 2.6 : Hypercube H(4)

L'*hypercube* de dimension p, H(p) est un graphe qui a 2^p sommets. Si l'on numérote ses sommets selon la représentation binaire des nombres de 0 à $2^p - 1$, deux sommets sont connectés si leurs étiquettes diffèrent d'un bit exactement. L'hypercube le plus utilisé est celui de dimension 4 (voir la figure 2.6).

Le degré de l'hypercube H(p) est p. La distance entre deux nœuds A et B d'un hypercube est égale aux nombre de bits différents entre A et B, soit la distance de Hamming $H(A, B)$.

Le diamètre de ce graphe est p et la distance moyenne est $p/2$. Le diamètre est calculé facilement par l'observation précédente sur les identités des sommets : par exemple pour aller du sommet $0 = 000\ldots0$ (base 2) au sommet $2^p - 1 = 111\ldots1$ (base 2), il faut mettre à 1 p bits, donc traverser p arêtes. La distance moyenne se calcule suivant les remarques suivantes :

• Le réseau est symétrique donc par exemple $\overline{d}_0 = \overline{d}$

•Pour aller du sommet 0 aux sommets ayant i bits à 1 il faut traverser i arêtes.

•Il y a $\binom{i}{p}$ sommets ayant i bits à 1.

•$\sum_{i=1}^{p} i \binom{i}{p} = p \sum_{i=1}^{p} \binom{i-1}{p-1} = p2^{p-1}$

2.2.7 Graphe de de Bruijn

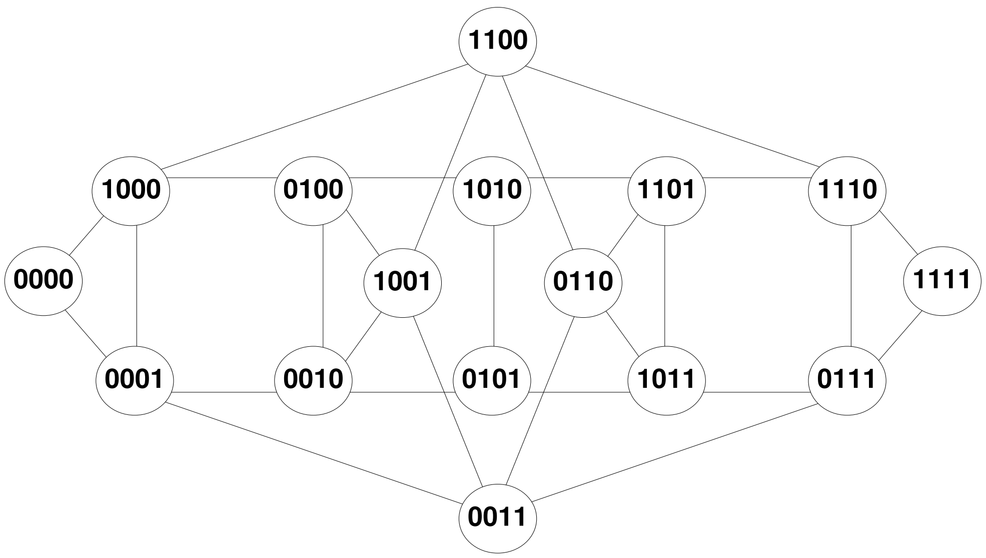

Figure 2.7 : Graphe de de Bruijn UB(2,4)

Un *graphe de de Bruijn UB(d,D)* est un graphe non orienté dont les nœuds sont des mots de longueur D dans un alphabet de d caractères [Brui46]. Il y a un lien de $(x_1, x_2, ...x_D)$ à tous les nœuds $(x_2, ..., x_D, \alpha)$ et $(\alpha, x_1, x_2, ..., x_{D-1})$, où α est un caractère quelconque de l'alphabet. Le passage direct d'un nœud à un autre correspond donc à un décalage (voir figure 2.7).

Le graphe UB(d,D) possède $n = d^D$ nœuds. Il n'est pas régulier puisque certains sommets sont de degré $2d$ et d'autres de degré $2d-2$.

Son diamètre vaut D puisque si l'on part d'un nœud quelconque on peut arriver à tout autre au bout de D décalages, toutefois, un algorithme de routage basé sur cette remarque donne une distance moyenne de D qui n'est pas optimale.

Ce type de graphe est très intéressant car, pour un même diamètre, les réseaux de de Bruijn comportent un plus grand nombre de sommets que les hypercubes qui sont pourtant beaucoup plus utilisés dans la pratique. Nous verrons par la suite que la grande influence des hypercubes sur les machines distribuées décline au profit de structures plus facilement extensibles (au sens augmentation d'échelle) comme la grille.

Pour une comparaison détaillée entre les graphes de de Bruijn et les hypercubes voir par exemple les travaux de J.-C. Bermond, C. Delorme, J.-J. Quisquater, A. Bouabdallah et J.-C. König, dans, entre autres, [BeDQ86, BoKo90].

2.3 Des machines

2.3.1 Introduction

Il y a loin entre l'IBM PC de nos débuts en informatique et les machines parallèles actuelles, nous avons pu ces 15 dernières années constater l'évolution des processeurs et des architectures. Dorénavant un processeur est en lui même une machine intrinsèquement parallèle, mais cela n'a pas suffit, l'enjeu est aujourd'hui de relier ces processeurs en d'immenses machines parallèles composées de milliers de ces processeurs. Ainsi la différence entre réseau et machine parallèle devient très faible pour ne pas dire inexistante, et il serait vain d'ignorer l'existence de ces machines pour ne considérer que des réseaux internationaux.

Des machines parallèles existent déjà, et leurs limites s'expriment par rapport à la mémoire disponible et aux délais de communication. Il semble maintenant évident pour de nombreux constructeurs que la solution réside dans des machines non plus parallèles mais en fait distribuées ; c'est à dire que pour faire face au besoin toujours croissant de puissance et de mémoire, une machine doit être *extensible* (ou *scalable* en anglais). Ce terme recouvre plusieurs notions dont la discussion sort des limites que nous nous sommes fixées, simplement nous considérons l'expression de la possibilité d'augmenter le nombre de processeurs (et la taille de la mémoire) sans avoir à changer toute l'architecture – toutes choses évidentes pour une machine sans mémoire commune donc distribuée.

Nous nous devons de remercier l'école d'automne CAPA [CAPA93] et particulièrement Michel Cosnard et F. Desprez pour leur cours sur les nouvelles machines parallèles.

Nous allons reprendre ici une partie de l'étude de M. Cosnard et F. Desprez [CAPA93] et y apporter nos remarques. Pour plus de détails sur les machines que nous allons brièvement présenter, le mieux est encore de consulter les manuels techniques des fournisseurs même si leur diffusion est assez limitée et sinon les quelques rapports de recherche et articles déjà disponibles sur le sujet. Nous ne détaillons pas les calculateurs T-Node de Telmat et MasPar MP-1 dont on trouve une étude à la fois précise et concise dans la thèse de Thierry Gayraud [Gayr92]. D'autres machines et des compléments sur celles présentées peuvent être trouvés dans le rapport de l'école d'été Rumeur [Rume93]. Pour un tour d'horizon assez complet sur les topologies, les machines et les processeurs le lecteur consultera avec intérêt la thèse de J. Sanchez [Sanc93].

2.3.2 Définitions usuelles

Toutes les machines disponibles peuvent se classer suivant les appellations donnés par la fameuse taxinomie de M. Flynn [Flyn66] : *SISD* Simple Instruction Simple Donnée soit « Single Instruction stream Single Data stream », *SIMD* Simple Instruction Multi Données soit « Single Instruction stream Multiple Data stream », *MISD* Multi Instructions Simple Donnée soit « Multiple Instruction stream Single Data stream » et *MIMD* pour Multi Instructions Multi Données soit « Multiple Instruction stream Multiple Data stream ». Le terme de « stream » utilisé ici fait référence au flot de données ou d'instructions vues par la machine pendant l'exécution d'un programme.

Une machine SISD est un ordinateur séquentiel classique. La CM-1 et la CM-2 sont des machines SIMD, il n'y a qu'un seul texte de programme et plusieurs données accessibles au même instant. La machine MISD n'a pas été très étudié, elle représente la composition de plusieurs unités de contrôle synchronisées, chaque unité commandant une unité de calcul. Cela correspond au schéma de type « pipeline »: le flux de données est unique, les unités de calculs traitent des parties distinctes de ce flux.

Nous ne considérons dans cette section que des machines MIMD, ce sont des machines à plusieurs processeurs ayant chacun au moins une mémoire privée. Les machines distribuées forment un sous-ensemble des machines MIMD, ce sont des machines sans mémoire commune.

Nous utilisons, pour tenter d'exprimer la puissance théorique de ces machines, le *Mips* Millions d'instructions par seconde et le *Mflops* Millions d'instruction flottantes par seconde qui fait référence à des opérations de manipulation de nombres réels (en l'absence de métrique plus perfectionnée car la puissance de calcul en MIPS et en MFLOPS est souvent discutée).

2.3.3 Thinking Machine CM-5

Thinking Machine Corporation a été le premier à concevoir des machines incluant plusieurs milliers de processeurs et même si ces processeurs était au départ très simples (à un bit), elles annonçaient les prémisses d'une course à la vitesse et au parallélisme massif. Les premières machines produites par cette firme (CM-1 et CM-2), étaient organisées sous forme d'hypercube, mais ce type de réseau est désormais abandonné (suivant en cela beaucoup d'autres constructeurs) au profit de topologies de type grille pour des raisons liés à des problèmes d'extensibilité. Entre autres raisons, il s'avère que pour garder la structure de l'hypercube, l'accroissement du nombre de nœuds implique l'accroissement des degrés de ces nœuds – ce qui est incompatible, ou tout du moins limitatif, au vu du matériel disponible.

La *CM-5* supporte les deux modèles de communication : SIMD et MIMD. Comme dans une machine MIMD, les processeurs peuvent en théorie exécuter des tâches différentes et ceci de façon asynchrone. Dans la pratique cette machine est plus couramment utilisée via ses capacités de machine SPMD synchrone avec les langages Fortran 90, C* et *Lisp. La CM-5 peut contenir en théorie jusqu'à 16 384 processeurs (16Ko) mais une telle machine occuperait une salle de 900 m^2 ! Une CM5 de 1 024 processeurs a été installée en juillet 1993 à Caltech, Pasadena, Californie.

2.3.3.a Un nœud de la CM-5

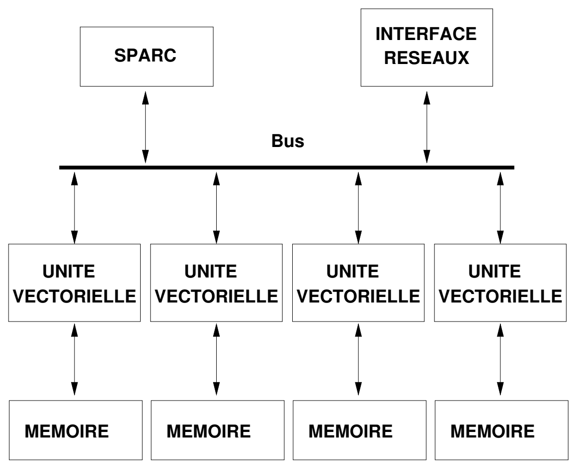

Figure 2.8 : Un nœud de la CM-5

Le processeur de base de la CM-5 est un processeur RISC (SPARC) auquel sont ajoutés quatre processeurs vectoriels 64 bits reliés par un bus de 64 bits pour former la « brique de base » de la CM-5 (voir la figure 2.8). Un bloc mémoire contient de 2 à 8 Mo. Ce type d'architecture permet en fait une utilisation de la CM-5 comme d'une machine à mémoire partagée.

2.3.3.b L'architecture de la CM-5

Cette machine se caractérise par un triple réseau de communication : un réseau de contrôle, de données et de diagnostic. Le premier permet l'utilisation d'opération globales synchrones sur plusieurs processeurs, le deuxième est utilisé par les communications asynchrones et le dernier, comme son nom l'indique, sert au dépistage des pannes et à la gestion des erreurs.

Techniquement parlant, les bandes passantes sur le réseau de données asynchrones sont de 20 Mo/s entre 2 processeurs voisins dans un groupe de 4. Dans les groupes de 16, la bande passante est divisée par deux et par quatre au delà de cette distance.

Chaque réseau est un Fat-Tree (voir le paragraphe 2.2.4), ce qui donne à ce réseau une résistance aux pannes par redondance. En effet, il est prévu au niveau matériel la possibilité de supprimer un processeur de communication de l'arbre (que l'on distingue des processeurs « utiles » qui sont au niveau des feuilles). Les nœuds de communication sont aussi à base de processeurs SPARC. Cette résistance aux pannes est limitée à une panne de ce que nous avons appelé les liens de communication du système distribué $\mathbb{S}$.

Les nœuds de la CM-5 peuvent être adressés par paquets par l'utilisateur ce qui peut faciliter la programmation de certaines applications.

2.3.3.c Les 3 systèmes de routage

Le système de routage du réseau de données est du type *worm hole* (littéralement trou de ver) c'est à dire que le système (matériel) de routage choisit un chemin de type profondeur d'abord avec choix aléatoire du successeur, l'utilisateur ne définissant que le nœud de départ et le nœud d'arrivée. Il est à noter que dans ce type de routage, les paquets de données arrivent de façon désordonnée, et donc le système doit gérer des copies des paquets et les ré-émettre si nécessaire.

Le système de routage du réseau de contrôle permet des opérations globales de type diffusion (synchrone), combinaison d'opérateurs globaux et des opérations de type **ou** sur 1 bit de façon synchrone ou non. Les opérateurs disponibles sont de type préfixe parallèle, suffixe parallèle, réduction et une opération spéciale de test de terminaison par message.

Le réseau de diagnostic est quant à lui inaccessible par l'utilisateur. Il n'est utilisé que par le système d'exploitation pour le contrôle du comportement des processeurs et leur éviction du calcul global si nécessaire (le système vérifie régulièrement le comportement des processeurs).

2.3.3.d Performances

Les processeurs SPARC ont une capacité théorique de 22 Mips ou de 5 Mflops. Les processeurs vectoriels fournissent 32 Mflops pour les multiplications et additions (car ils peuvent faire une addition et une multiplication en un même cycle) tandis que les opérations d'accès à la mémoire sont au mieux de 16Mflops. De plus amples détails sont disponibles dans les articles et rapports de Leiserson et Palmer [Leis85, LADF+93, Palm92].

2.3.4 La KSR1

Conçue par Kendall Square Research en 1991, elle est basée sur une hiérarchie d'anneaux. En fait, un anneau d'anneaux, pour les versions les plus courantes. Pour l'utilisateur, elle a l'avantage de fournir une *mémoire partagée virtuelle*, c'est à dire que la mémoire peut être vue comme une seule mémoire unique commune à tous les processeurs – ce qui en facilite la programmation. Toutefois, cette machine est réellement une machine distribuée et, même si cette réalité est en grande partie cachée à l'utilisateur, une réelle optimisation des programmes passe toujours par une réécriture du code en ayant à l'esprit que les temps d'accès à la mémoire non locale sont d'un coût non négligeable.

2.3.4.a Un nœud de la KSR1

Un nœud est composé de 12 circuits CMOS fabriqués par KSR. Ils sont répartis comme suit :

- Une unité de distribution : lecture/écriture des données, calcul d'adresses, contrôle du flot d'exécution...
- Une unité de calcul entier : opérations entières et logiques.
- Une unité de calcul flottant.
- Une unité d'entrée/sortie.
- 4 unités de contrôle de cache : interface entre mémoire locale (32 Mo) et des sous-caches de transfert.
- 4 unités d'interconnexion entre un nœud de calcul et l'anneau principal.

2.3.4.b L'architecture de la KSR1

La *KSR1* est basée sur une hiérarchie d'anneaux de 32 nœuds auxquels s'ajoutent deux cellules responsables du routage avec l'anneau supérieur. L'architecture théorique maximale est une hiérarchie en 3 niveaux : un anneau d'anneaux d'anneaux, comme nous le présente la figure 2.9. Quelques intéressantes expérimentations sur cette machine peuvent être trouvées par exemple dans le rapport de T. Dunigan [Duni92].

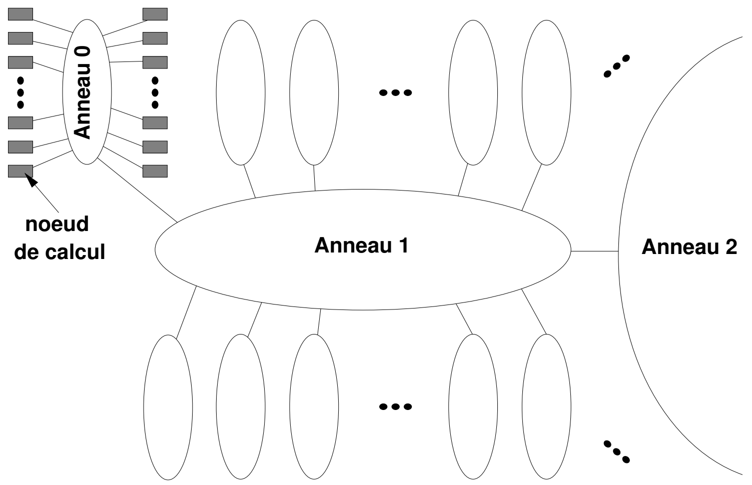

Figure 2.9 : Architecture de la KSR1

2.3.4.c Performances

Le constructeur donne une puissance théorique de 40 Mflops par nœud. Il ne nous a pas été possible d'atteindre de telles performances et plus particulièrement nous n'avons pas pu constater, dans nos premiers essais, d'accélération significative par rapport à une machine à 10 processeurs et mémoire partagée de conception déjà ancienne : la Sequent Balance B8000.

Peut être faut-il incriminer l'instabilité de certaines fonctions de la librairie C parallèle (*getsubpage()* par exemple dans la version datée du 30 juillet 1993) qui conduisent parfois à des erreurs fatales ! Ces quelques anomalies ne sauraient cacher les intérêts réels de cette machine qui s'avère donner de très bons résultats sur des applications de grande taille utilisant essentiellement des opérations de manipulation de matrices et très peu de verrous (que ce soit exclusion mutuelle ou barrières de synchronisation).

2.3.5 La Paragon

Chez Intel Corporation, la machine Paragon fait suite à la machine Delta qui n'a pas été réellement commercialisée. Elle en reprend les concepts principaux mais avec deux processeurs par nœud et un nouveau système d'exploitation. Ici encore, la stratégie initiale avec une architecture en hypercube (la machine iPSC/860) a été abandonnée au profit d'une autre architecture pour des raisons d'extensibilité. La grille a été retenue pour la Paragon (car simple à fabriquer et de routage aisé), avec un nombre maximum de nœuds fixé à 2 048.

2.3.5.a Un nœud de la Paragon

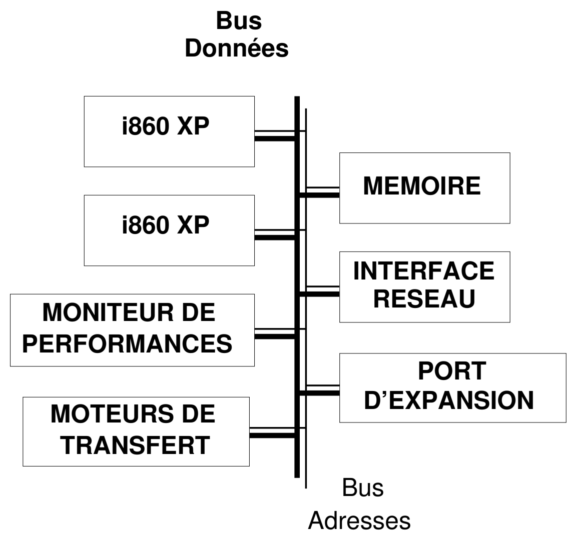

Figure 2.10 : Un nœud de la Paragon

Un nœud de la Paragon comporte deux bus (un bus de données de 64 bits et un bus d'adresse de 32 bits), deux processeurs i860XP, de 16 à 64 Mo de mémoire et différents éléments précisés par la figure 2.10. Les deux processeurs sont utilisés de façon parfaitement distincte : le premier sert à l'application et le deuxième est entièrement dédié à la communication, ils travaillent en parallèle sur la même mémoire. Pour d'amples détails sur le processeur i860XP voir l'article de S. Margulis dans [Marg89].

2.3.5.b L'architecture de la Paragon

Le processeur i860XP ne dispose pas de liens avec l'extérieur c'est pourquoi tout nœud de la Paragon est lié avec un processeur spécialisé de communication : le iMRC (Mesh Routing Chip) comme illustré sur la figure 2.11.

Le système contient trois types de nœuds : les nœuds de calcul, de service et des nœuds d'entrée sortie. Les nœuds de services fournissent la possibilité d'utiliser le système d'exploitation UNIX incluant la compilation et les outils de

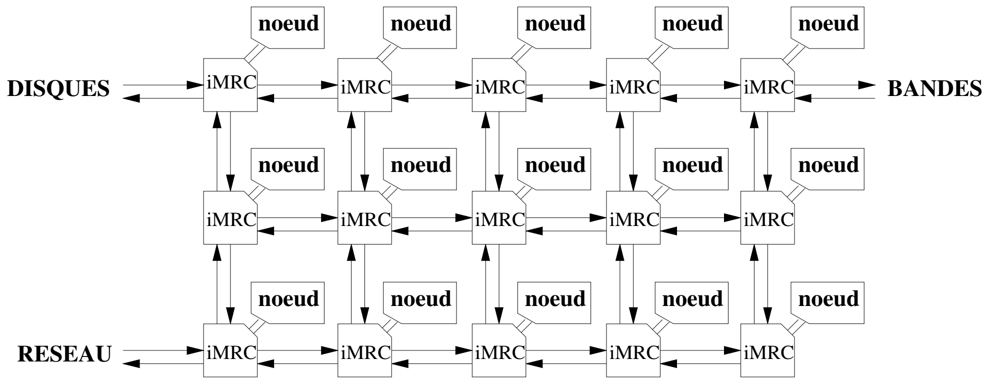

Figure 2.11 : Architecture de la Paragon

développement de programmes. Les nœuds d'Entrée/Sortie font l'interface entre les autres nœuds et la mémoire de masse. Il faut signaler aussi la présence d'un réseau de diagnostic dédié à l'initialisation de la machine et aux diagnostics.

2.3.5.c Performances

En théorie, avec 2 048 nœuds de calcul, la Paragon peut atteindre les 130 Gigaflops, la bande passante étant estimée à 200 Mo/s. En pratique, les résultats obtenus par T. Dunigan dans [Duni92] sont très intéressants et prouvent, au moins pour des problèmes de type manipulation de matrice, que cette machine permet à la fois un portage relativement aisé et l'obtention de bonnes performances.

2.4 Rappels sur le modèle OSI

2.4.1 Introduction

La normalisation de l'Interconnexion des Systèmes Ouverts OSI (Open Systemes Interconnection) est un processus engagé depuis 1977. Cette normalisation internationale est confiée à l'ISO (International Standards Organization). Les membres de cette organisation sont des associations nationales de normalisation, une par pays. Pour la France, L'AFNOR (Association Française de Normalisation) est l'unique organisme faisant l'interface avec l'ISO.

Nous allons rappeler les fameuses 7 couches de l'OSI en essayant de distinguer ce qui est du ressort de la théorie et ce que peut être la pratique. Pour plus de précisions sur le sujet, les ouvrages de A. Tanenbaum et P. Rolin [Tane90, Roli89] qui font référence. On pourra consulter aussi l'article de J.-F. Gornet et M. Levilion [GoLe91] sur un état de la normalisation.

2.4.2 Description synthétique

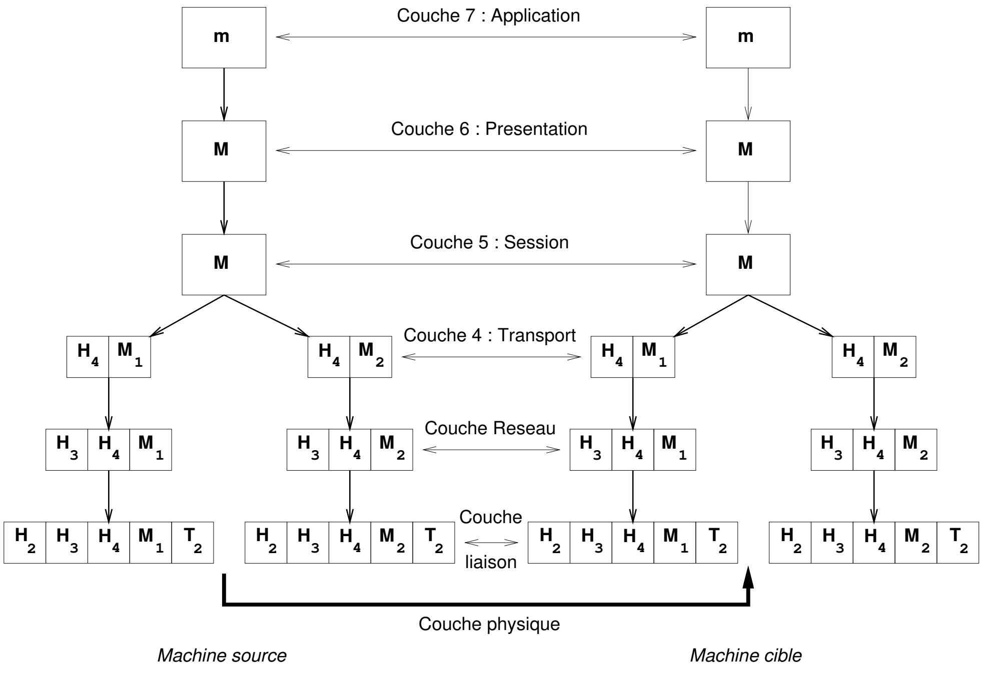

Figure 2.12 : Les 7 couches de la norme OSI

La figure 2.12 représente le chemin que suit l'émission d'un message m d'une machine à une autre. Le message m est produit par un processus au niveau 7, le message est passé de la couche 7 à la couche 6 via l'interface (la flèche verticale sur la figure) 6/7 – il y a alors transformation du message (compression de texte par exemple) en un message M. Ce message est passé à la couche 5 suivant les règles régissant l'interface 5/6. Dans l'exemple de la figure, l'interface 5/6 et la couche 5 n'entraine pas de modification de message.

Dans la plupart des réseaux, il n'y a pas de limite imposée sur la taille des messages acceptés par la couche 4, cependant, due à la limite imposée au niveau 3, la couche 4 découpe le message en M_1 et M_2. A chaque fragment de message (paquet) on associe une en-tête (H_4 ici), celle-ci contient des informations de contrôle telle que l'ordre du paquet dans la séquence des paquets, de sorte que la machine cible puisse remettre dans le bon ordre les fragments du message. Cette information est importante car certains protocoles de routage répartissent les paquets suivant des chemins pas forcément tous identiques ce qui induit un déséquencement à l'arrivée. Dans la plupart des couches, les en-têtes contiennent des tailles, numéros de séquence (dates) et autres champs de contrôle.

La couche 3 décide quelle ligne de communication choisir, attache ses propres entêtes et passe les paquets (nommés alors trames) à la couche 2. Celle-ci ajoute non seulement sa propre en-tête mais aussi une postface : c'est ce que l'on appelle l'encapsulation. Enfin, la couche 2 envoie le message sur le niveau physique. Sur la machine cible, le message est réassemblé et décodé dans l'ordre inverse avec comme règle absolue qu'aucune en-tête des couches inférieures à c n'est transmise à la couche c.

2.4.3 En pratique ?

En résumé, la couche 7 est la couche sur laquelle tournent les utilitaires classiques type courrier électronique, gestion des transferts de fichiers, sessions distantes, etc.

Les fonctions majeures de la couche 6 sont le codage et la compression des données ainsi que la conversion de code (par exemple la conversion de ASCII à EBCDIC).

Les couches 5 est plus particulièrement liée à la sémantique des communications (par exemple l'appel de procédure à distance dans l'Internet ou mode message pour l'OSI).

La couche 4 assure la reprise sur erreur, le séquencement, le multiplexage, le contrôle de flux... La couche 3 effectue les opérations liés à la segmentation, et le routage. Quant à la couche 2, elle est fortement liée au réseau, elle contient par exemple les protocoles de gestion des trames dans la norme Ethernet. Elle doit effectuer la détection et la correction éventuelle des erreurs par des codes polynomiaux de contrôle (test de parité par exemple).

Le modèle OSI, comme tous les autres, à été défini alors que déjà des solutions existait. Ces solutions ont alors été adaptées pour pouvoir respecter le schéma en 7 couches. Rares sont les applications réelles utilisant par exemple la couche 6 et nombreuses sont celles faisant l'amalgame des couches 3 à 5.

D'autres développements et remises en cause ont encore lieu, la norme tente de suivre l'évolution des techniques et technologies qui en vingt ans ont évidemment bien changé. L'arrivée des systèmes répartis est certainement l'occasion d'harmoniser et de moderniser les normes sur les réseaux.

2.5 Classification des pannes

Un réseau en fonctionnement peut subir différentes atteintes imprévisibles. Que les atteintes proviennent des rongeurs qui mangent les câbles[2], de la foudre ou autres phénomènes naturels ou encore, d'un point de vue militaire, espionnage ou autre, que des malveillances soient commises en vue de la destruction du réseau, les causes de pannes sont rarement évitables et tout doit être fait pour tendre vers une fiabilité maximale.

Il est à présent nécessaire de distinguer système distribué et algorithme distribué. Nous avons présenté dans le chapitre précédent le modèle de système distribué asynchrone $\mathbb{S}$ ainsi que les algorithmes distribués asynchrones $\mathbb{A}$ fonctionnant sur ces systèmes. Pour détecter une panne, un système distribué se doit de pouvoir connaître des bornes sur les délais de communication car l'on ne peut distinguer un site en panne d'un site très lent (voir M. J. Fischer *et al.* [FiLP85]). Autrement dit, le modèle asynchrone ne peut permettre la prise en compte des pannes. Il y a lieu ici d'étendre ce modèle vers une forme de synchronisation nécessaire : la connaissance de bornes supérieures (pour tout lien) des délais de communication. Notons $\mathbb{S}'$ le modèle obtenu. Sur $\mathbb{S}'$, il est toujours possible de faire tourner $\mathbb{A}$, de fait nous n'avons besoin que de la détection de l'erreur, qui est gérée au niveau du système, pour pouvoir décrire des algorithmes asynchrones résistants aux pannes.

Dans les paragraphes suivants, nous allons définir une partie du vocabulaire communément utilisé dans le domaine des pannes. D'autres termes existent, mais

[2] Pour l'anecdote, les lapins à l'Inria avaient trouvé à leur goût le matériau enrobant des câbles inter-bâtiments, ce qui a obligé le constructeur des câbles à changer de technique de blindage !

le vocabulaire présenté ici nous est déjà apparu suffisant pour que nous nous restreignons à l'emploi d'une partie seulement de l'ensemble des terminologies utilisées dans ce domaine (voir entres autres J.-C. Laprie [AnLe81, Lapr92]).

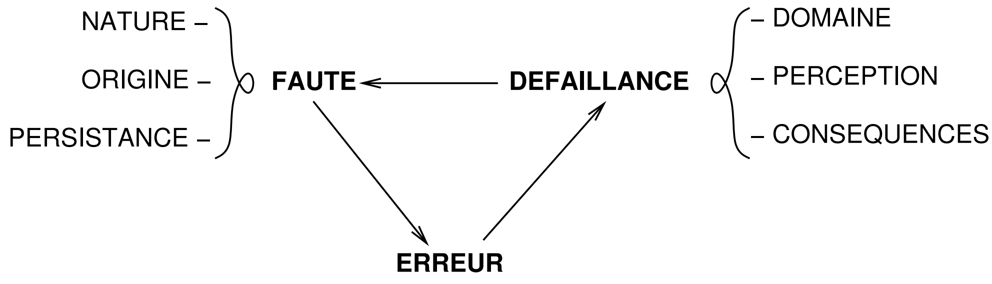

Figure 2.13 : Classification des pannes

2.5.1 Les fautes

Toute *faute* comporte trois caractéristiques : sa *nature*, son *origine* et sa *persistance*. C'est presque une lapalissade de dire qu'une faute est de nature soit accidentelle soit criminelle, mais son origine doit être classée selon un des deux types suivants :

- cause «phénoménologique» (phénomènes physiques, fautes humaines, etc.)
- phase de création (fautes de conception, fautes dues à des modifications ultérieures, fautes opérationnelles durant l'exploitation du réseau).

Reste le paramètre de persistance temporelle qui tente de distinguer les fautes *permanentes* des fautes *temporaires* que l'on subdivise généralement en fautes *transitoires* (fautes externes) et en fautes *intermittentes* (fautes internes).

Enfin une faute peut entraîner ou non une *erreur* suivant la fonction assurée par le processus au sein du système et l'extension éventuelle de cette faute au système entier (en particulier, dans les réseaux, selon les protocoles assurant le transport des messages dans le réseau).

2.5.2 Les erreurs

Une erreur est un état susceptible de conduire à une *défaillance*, tandis que la faute est la cause supposée ou adjugée d'un erreur. Des défaillances peuvent d'ailleurs entraîner des fautes qui elles-mêmes peuvent entraîner des erreurs, etc...

L'ensemble de ces interactions se résume par la figure 2.13 où les arcs symbolisent les conséquences possibles.

2.5.3 Les défaillances

La *défaillance* est la vision qu'a l'utilisateur de l'erreur. Les défaillances se classent suivant trois critères : le *domaine*, la *perception* par les utilisateurs du système de cette défaillance et enfin ses *conséquences*. Les subdivisions caractéristiques de chacun de ces critères sont résumées comme suit :

- Parmi les domaines de défaillance, le *domaine des valeurs* (non conformité par rapport aux spécifications) et le *domaine temporel* (le message arrive trop tôt ou trop tard)[3].
- La perception de la défaillance peut être *cohérente* (perçue par tous les utilisateurs de façon identique) ou *incohérente* (l'incohérence peut porter sur la perception en elle-même ou sur le fait que tous ne constatent pas la défaillance).
- Enfin, la défaillance est plus ou moins grave : de *bénigne* à *catastrophique*, ce facteur est souvent lié à un problème économique : toute heure de non-disponibilité peut coûter des milliards de francs à certaines entreprises (gestion boursière par exemple), alors que d'autres peuvent supporter un arrêt de plusieurs heures sans trop de gène.

Il existe une vision ensembliste des défaillances proposée par F. Cristian, H. Aghili, R. Strong et D. Dolev dans [CASD85] : l'arrêt sur défaillance est un cas particulier de la défaillance par omission, elle même incluse dans la défaillance temporelle (dans un réseau à délai de transmission borné). Le cas le plus général étant la défaillance arbitraire ou byzantine englobant toutes les autres en plus de la possibilité de falsification des messages par des sites. Cette défaillance a donné lieu au problème dit des généraux byzantins (voir entre autres l'article de L. Lamport *et al.* [LaSP82]).

2.6 Résistance aux pannes

Si une erreur peut être détectée avant la défaillance, alors le réseau sera résistant à ce type de panne.

Nous réduisons notre étude aux défaillances par arrêt suffisamment long pour être détectable. C'est à dire que des fautes diverses entraînent des erreurs causant

[3] Ce dernier cas ne nous concerne pas d'après les hypothèses faites en 1.4.5, c'est le propos du Temps Réel

la défaillance par arrêt (coupure) de liens de communication ou de sites. Par extension, nous étudions aussi l'ajout de liens de communications en cours de fonctionnement. Ces pannes seront appelées pannes franches par opposition aux pannes aléatoires ou transitoires.

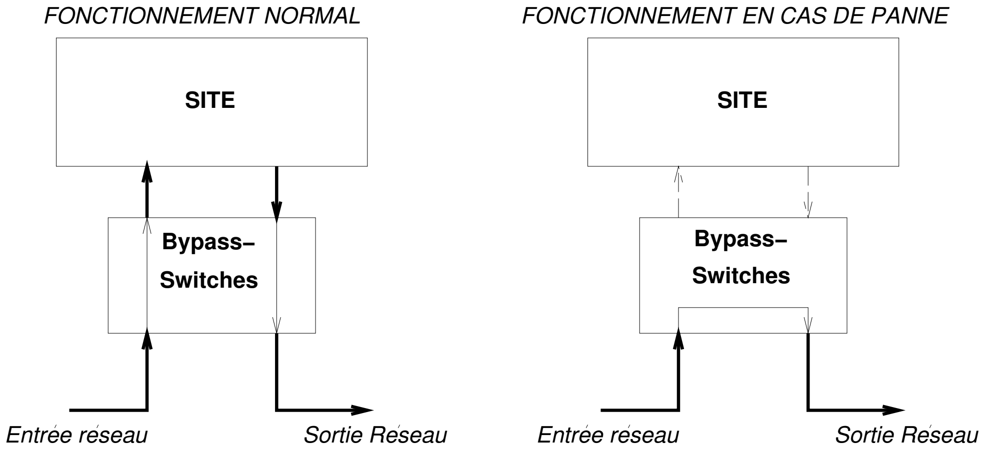

Figure 2.14 : Représentation schématique des « bypass-switches »

Il est à noter que techniquement une panne (franche) de ligne peut souvent se détecter par les sites extrémités de cette ligne, que ce soit par chute de porteuse ou autres techniques de bas niveau. De même, il est techniquement possible, par l'utilisation de « ponts » (« bypass-switches », voir figure 2.14) à bascule mécanique (débranchement) ou électronique (bascule commandée ou « automatique » par dialogue local avec le site), d'assurer qu'en cas de panne (franche) de site, les messages destinés à ce site ne soient pas perdus mais retransmis sur d'autres lignes ce qui permet alors au site receveur de détecter la panne du site qui aurait normalement dû recevoir le message. En effet, il est possible pour un site d'associer, dès qu'il a reçu au moins un message d'un voisin x, le numéro de porte par lequel il a reçu le dernier message de x et l'identité de x. Cette technique est parfois suppléée par un protocole de test régulier de la présence active des voisins (messages de type « êtes-vous là ? »).

Cette dernière doit être associée à la connaissance de bornes sur les délais de transmission contrairement au modèle de système asynchrone (voir la section 1.4). Ces bornes peuvent être uniquement locales dans le cas du dialogue local entre le site et le « bypass-switch ». Reste alors a ce dernier à ne pas tomber en panne !

La panne franche d'un processeur peut se considérer suivant différents point de vue. Le premier considère que la panne est indétectable et dans ce cas aucun algorithme distribué asynchrone ne peut établir de consensus (par exemple l'élection) sur l'ensemble du réseau car l'on ne peut distinguer un site en panne d'un site très lent (voir M. J. Fischer *et al.* [FiLP85]). Le deuxième point de vue consiste comme nous l'avons déjà décrit, de supposer que les messages ne sont pas absorbés par le site défaillant mais qu'ils continuent leur route vers un autre voisin ce qui permettra de détecter la panne. Enfin, il est d'usage de représenter la panne d'un site comme la panne de toutes les lignes de communication menant vers ce site, ainsi on pourra détecter cette panne puisque l'on fait l'hypothèse que l'on peut détecter les pannes de lignes dans le modèle $\mathbb{S}'$.

Nos hypothèses nous placent dans un environnement ou les pannes sont détectables, donc, si le graphe reste connexe, au bout d'un temps fini après la dernière panne, nous pourrons assurer dans certains cas, par des techniques de reprise ou de tolérance aux pannes, la bonne exécution de l'algorithme. Nous utilisons en fait un point de vue optimiste, c'est à dire que les pannes ne surviennent que rarement et les algorithmes que nous considérons n'entraînent pas de surcoût pour la gestion spécifique de la tolérance aux pannes lorsqu'aucune panne ne survient.

Pour des définitions et discussions plus complètes sur la sûreté de fonctionnement, voir l'article de J.-C. Laprie [Lapr92] et, pour un tour d'horizon des problèmes insolubles de l'algorithmique distribuée, N. Lynch [Lync89].

2.7 Conclusion du chapitre 2

Les limitations technologiques se ressentent fortement sur le degré des réseaux, ainsi un réseau de processeurs TMS320C40 [TEXAS92] ne peut dépasser un degré de 6 et un réseau de transputers T800 [INMOS87] un degré de 4. D'une manière plus générale, le degré d'un réseau réel est souvent borné par une constante.

Néanmoins nous étudions différents types de réseaux et les algorithmes, que nous présentons dans le chapitre suivant, considèrent toujours la topologie comme quelconque car un réseau réel en fonctionnement est un réseau dont la topologie est variable (incidents divers sur les liens de communications ou sur les sites) et dont on ne connaît pas, à un instant donné, ni la topologie ni le nombre de nœuds. De ce fait, la reprise sur pannes, peut conduire à construire une structure de contrôle sur une topologie que l'on ne connaît pas à l'avance.

De plus il est souvent inutile, voir limitatif, de supposer l'existence d'un maître ou élu prévalent à l'exécution des algorithmes distribués, l'extensibilité des machines ne saurait être limitée par de telles hypothèses. Les algorithmes utilisant l'hypothèse d'un initiateur ou d'un site privilégié avant toute exécution n'auront fatalement qu'une résistance aux pannes très limitée car une panne de ce maître est catastrophique.

Chapitre 3

Algorithmes de contrôle distribués (Arbres Couvrants)

3.1 Introduction

Nous allons étudier dans ce chapitre différentes structures de contrôle distribuées couvrant la totalité du graphe. La littérature sur le sujet nous renvoie essentiellement à la construction d'anneau (ou cycle) hamiltonien (ou pseudo-hamiltonien) ou d'arbre couvrant. Dans tous les cas, il faut souligner la volonté de briser la symétrie d'origine et d'organiser le réseau en structure cohérente.

Un anneau hamiltonien est de construction difficile (le problème est NP-complet) et il existe des graphes (les arbres par exemple) sur lesquels on ne peut pas construire des anneaux hamiltoniens. D'où l'idée de l'anneau pseudo-hamiltonien qui est un anneau passant au moins une fois par chaque sommet (au lieu d'une et une seule fois). Les défauts d'une telle structure sont connus : très faible potentiel de résistance aux pannes (certains sites étant vus plusieurs fois), délais de communication plus longs en moyenne que pour les arbres etc. De plus la construction d'un anneau virtuel utilise souvent l'algorithme de parcours en profondeur (voir par exemple [HeRa87]), par conséquent, soit l'on dispose déjà d'un seul initiateur soit il faut en même temps établir une élection particulière. En clair, l'élection sur un anneau virtuel consiste à faire une élection, la création de l'anneau et ensuite l'élection sur cet anneau !

Un des avantages de la topologie en anneau est que de nombreux algorithmes optimaux à la fois en temps et en messages existent déjà (voir par exemple l'algorithme de E. J. Chang et R. Roberts [ChRo79]).

L'anneau pseudo-hamiltonien ou anneau virtuel à été étudié, entre autres, par I. Lavallée qui propose un algorithme de construction simple par jeton circulant dans [Laval90].

La construction d'un arbre couvrant n'apporte pas seulement une structure de contrôle sur la totalité du réseau, mais aussi, un élu. En effet, le problème de l'élection, de la recherche d'extremum et de la terminaison sont réductibles au problème de la construction d'un arbre couvrant comme nous le prouve l'analyse de la section 3.1.2.

Nous ne nous intéressons qu'à l'étude des arbres couvrants avec ou sans contraintes. Ce problème, pour la contrainte de poids total minimum ou dans le cas d'absence de contrainte, a été largement étudié et de nombreux algorithmes, tant séquentiels que distribués, ont été proposés (voir par exemple [Awer87, Bute93, GaHS83, GrHe85, KoKM90, LaLa89a]). Nous nous intéressons ici à une contrainte importante, qui n'a jamais, à notre connaissance, été étudiée en algorithmique distribuée : le diamètre minimum. L'obtention d'une structure de contrôle de diamètre minimum est d'un intérêt évident puisqu'il existe toujours au moins une paire de sites qui doit, pour échanger des informations, traverser D arêtes où D est le diamètre de la structure de contrôle utilisée. Nous présentons dans ce chapitre un algorithme distribué de construction d'arbre couvrant de diamètre minimum sur le réseau de communication.

On examine donc différents algorithmes sous différents points de vue. Tous les auteurs cités cherchent « l'économie » en nombre de messages, mais pas tous de la même façon.

En effet, on retrouve ici, sous une autre forme, les termes du compromis temps-espace bien connu en algorithmique et programmation séquentielle. On peut diminuer la quantité de mémoire nécessaire à une application mais cela représente souvent du temps perdu et, inversement, un gain de temps est souvent obtenu au détriment de l'espace mémoire utilisé. Ici, nous sommes confrontés à un problème analogue, en limitant le parallélisme des opérations dans le réseau, on arrive à limiter de façon drastique le nombre de messages échangés dans le pire des cas [Awer87, GaHS83, KoKM90]. Toutefois, pour ce faire, les auteurs sont obligés d'utiliser la notion de phase logique qui induit des phénomènes d'attente. De plus, la résistance aux pannes peut s'en trouver affaiblie.

Partant de l'idée que le pire des cas n'arrive pratiquement jamais, d'autres auteurs tels que [Bute93, LaLa89a, LaRo86], ont proposé des algorithmes qui sont totalement asynchrones, dynamiques et ne nécessitant que peu d'attentes. Ils souffrent d'un peu plus de messages échangés et en théorie ils peuvent sur des exemples « pathologiques » avoir des comportements « gourmands » en nombre de messages, mais dans la pratique, ils restent du même ordre comme nous le prouve le chapitre 4.

Nous proposons de plus des techniques d'amélioration des algorithmes présentés pour obtenir une certaine résistance aux pannes.

3.1.1 Notations

Nous reprenons les définitions données en section 1.2 et ajoutons les notations suivantes :

Soit $N_{mot\_clé}$ le nombre total de messages portant le mot-clé *mot_clé* ayant circulé dans le réseau pour obtenir la terminaison de l'algorithme.

Dans tous les codes des algorithmes fournis par la suite, la constante EGO désigne l'identité du site sur lequel est décrit l'algorithme. De même, la constante VOISINS désigne l'ensemble des voisins immédiats du site considéré. Il serait plus juste par rapport au modèle $\mathbb{S}$ de n'utiliser que l'ensemble des numéros de porte vers les voisins (puisque leurs identités sont *a priori* inconnues) mais il s'agit ici que d'une notation destinée à simplifier l'écriture des algorithmes.

3.1.2 Élection et construction d'arbre couvrant

Le problème de l'*élection* consiste, dans un système distribué $\mathbb{S}$, à une transformation de la configuration de $\mathbb{S}$: d'une configuration initiale du système où tous les sites actifs sont dans le même état fondamental que l'on appelle *candidat*, à la configuration finale où un seul site est dans un autre état fondamental appelé *élu* et tous les autres sont dans l'état *battu*. L'élu est *a priori* quelconque, bien que la plupart des algorithmes d'élection, sur des réseaux dont les sites disposent d'identités toutes distinctes, transforment ce problème en la recherche d'extremum sur l'ensemble des identités des sites et élisent cet extremum comme le fait, par exemple, le premier algorithme connu d'élection (G. Le Lann [Lela77]).

De même, les algorithmes de construction d'arbre couvrant utilisent, lorsque cette information est disponible, les identités des sites pour résoudre les problèmes de conflit. Ces identités fournissent généralement aussi les identités des fragments qui vont chercher à se fusionner. De sorte que le site ayant la plus grande (ou la petite) identité joue un rôle primordial. Nous avons, dans notre algorithme de construction d'arbre couvrant, cherché à limiter ce rôle de l'extremum, de même l'algorithme de E. Korach *et al.* dans [KoKM90] utilise l'ordre induit sur les identités des sites mais ne privilégie pas d'extremum.

3.1.2.a De l'élection à l'arbre couvrant

Si l'on dispose d'un élu, un site d'identité x, il est possible de procéder à partir de cet élu à un parcours du réseau en largeur d'abord qui nous donne alors une arborescence (logique) couvrante de racine x. L'analyse de l'algorithme **BFS** suivant va nous montrer que la *transformation* du problème de l'élection en problème de construction d'un Arbre Couvrant (ou plutôt d'arborescence couvrante) peut s'établir via l'échange de $2m$ messages et de l'ordre de $2D$ unités de temps.

3.1.2.b Algorithme BFS

Algorithme **Distri-BFS**

```
premier_message←VRAI ;
SI élu = EGO ALORS
   fils←VOISINS ; attente←|fils| ;
   premier_message←FAUX ;
   ENVOIE <bfs> à ses fils ;
FSI
TANT QUE mon_état ≠ terminé FAIRE
   Sur réception de message <bfs> de y :
                        (* Seul le 1er mesg. reçu définit l'arborescence *)
      SI premier_message ALORS
         premier_message←FAUX ; père← y ; fils←VOISINS−{y} ;
         ENVOIE <bfs> à ses fils ;
         attente←|fils| ;
      SINON                      (* « Élimination » de l'arête (EGO, y). *)
         fils←fils−{y} ; attente←attente−1 ; (* Pas de mesg. renvoyé *)
      FSI
   Sur réception de message <bfs_back> :
      attente←attente−1 ;
   SI attente=0 ET NON premier_message ALORS
      SI élu = EGO ALORS
         mon_état← terminé ;           (* l'algorithme est terminé. *)
      SINON
         ENVOIE <bfs_back> à son père ;
         mon_état← terminé ;                  (* localement termin
      FSI
FIN TANT QUE.
```

Lemme 5 *Durant toute exécution de l'algorithme* **Distri-BFS**, *il y a* $2m$ *mes-*

sages de un bit échangés.

Preuve : Cet algorithme procède par «inondation» du graphe à partir de l'élu x. Cet algorithme distribué n'a donc qu'un seul initiateur : x.

Les messages sont uniquement de deux types : ***bfs*** et ***bfs_back*** que l'on peut coder respectivement par 0 et 1, donc les messages ne sont que de 1 bit. Lorsque nous faisons référence à l'émetteur du message il ne s'agit que d'une simplification d'écriture (voir paragraphe 1.4.5) ; en fait le message est reçu par la porte p, qui accessoirement mène vers y mais cette information n'est pas ici nécessaire, donc l'identité de l'émetteur n'est pas portée par le message.

Les messages de type ***bfs*** parcourent tout le graphe, plus précisément, toutes les arêtes du graphe. Une arête sur laquelle deux messages de ce type se croisent est dite «éliminée». Cette élimination est effectuée par les sites extrémités de cette arête via la suppression dans l'ensemble fils du voisin correspondant.

Le nombre de messages de type ***bfs*** est égal à $m + a$, où m est le nombre d'arêtes du graphe et a le nombre d'arêtes comptées deux fois : les arêtes éliminées. Les messages ***bfs_back*** ne circulent que sur les arêtes non éliminées donc $m - a$ messages de ce type sont émis durant l'exécution de l'algorithme. Ce qui donne au total les $2m$ messages annoncés. □

Lemme 6 *L'algorithme* **BFS** *construit un Arbre couvrant.*

Preuve : Soit $G' = (X', U')$ le graphe G auquel on a ôté les arêtes éliminées. Ce graphe est tel que $X' \subset X$, $U' \subset U$, $|U'| = m - a$ et $x \in X'$. L'algorithme ne génère pas de cycle car il ne peut exister deux chaînes de la racine vers un nœud (de par la gestion de la variable premier_message).

De plus, puisque G est connexe, tous les nœuds de G sont visités donc clairement $X' = X$. Par conséquent, d'après les définitions du paragraphe 1.2.2, G' est un Arbre Couvrant. □

Remarques

-La gestion des variables père et fils conduit en fait l'algorithme à construire une arborescence couvrante.

-Le nombre a d'arêtes éliminées est égal à $m - |U'|$ or G' est un arbre donc $|U'| = n - 1$ et $a = m - n + 1$.

Pour l'analyse de la complexité temporelle d'un algorithme, nous devons fixer une borne sur les délais de transmission. Uniquement à des fins d'analyse, supposons donc que l'algorithme est exécuté sur un réseau synchrone, il faut alors, pour l'exécution complète de l'algorithme, que les messages se propagent jusqu'aux sites les plus éloignés de l'élu et en reviennent, ce qui signifie dans le pire des cas $O(D)$ unités de temps, où D est le diamètre du graphe.

Remarque : On peut simplifier à l'extrême cet algorithme en n'utilisant pas de variable attente et en émettant le message ***bfs_bak*** dès sa réception ou si l'ensemble des fils est vide.

Dans ce nouvel algorithme, il est possible que des sites dans un état terminé reçoivent encore des messages qui seront alors ignorés. L'élu s'arrête dès le premier message ***bfs_back*** reçu alors que certains de ses fils (et petit-fils) n'ont pas encore terminé, mais tous, au bout d'un temps fini (en $O(D)$) finissent par terminer.

Pour obtenir un comportement plus « propre » de l'algorithme, il faut qu'un site attende que tous ses fils lui aient envoyé un message ***bfs_back*** pour qu'il puisse à son tour envoyer ce message vers son père. C'est la fonction de la variable attente.

Dans une terminaison par processus, il est nécessaire qu'un site qui a terminé l'exécution de l'algorithme ne reçoive plus de messages ayant un rapport avec cet algorithme. Dans le cas contraire, il pourrait être réveillé par le message et ne pourrait alors déterminer si le message fait partie de cet algorithme ou d'une nouvelle exécution.

Du point de vue de la complexité temporelle, attendre que tous ses fils aient terminé, ou bien terminer dès que l'un d'entre eux ait terminé, n'apporte pas de changement d'ordre de grandeur pour ce type d'algorithme.

En fait, si l'on suppose que la fonction de poids w définie sur les arêtes du graphe représente un délai moyen de propagation de message en unités de temps arbitraires et si ce délai reste relativement stable durant l'exécution de l'algorithme, alors il faudra au plus $2e(x) + W$ unités de temps pour compléter l'exécution de cet algorithme. $e(x)$ a déjà été définie dans le chapitre 1 et représente l'excentricité du nœud x ; W représente le poids de l'arête la plus « longue » qui ait été éliminée, c'est à dire celle de poids associé le plus grand (voir la figure 3.1).

Théorème 2 *Si la fonction de poids $w(e)$ représente exactement le délai de communication de toute arête e, et si ce délai reste parfaitement stable pendant l'exécution de l'algorithme, alors l'algorithme* **BFS** *construit un Arbre des Plus Courts Chemins issus de l'initiateur x (avec $2m$ messages).*

Preuve : Nous avons déjà vu, par les lemmes précédents, que l'algorithme **BFS** construisait un arbre couvrant avec $2m$ messages échangés.

Considérons maintenant un nœud $y \neq x$, supposons qu'il existe au moins deux chemins μ_1 et μ_2 issus de x et tels que $d(x, y) = w(\mu_1) = w(\mu_2)$. Si, comme le spécifie le théorème, la fonction de poids représente exactement le délai de communication, et que ce délai reste parfaitement stable, deux messages ***bfs*** issus de x rejoindront y simultanément, au bout de $d(x, y)$ unités de temps. L'algorithme opère alors un choix arbitraire sur l'ordre d'interprétation de ces deux messages comme spécifié dans le paragraphe 1.4.5 et propage alors le message

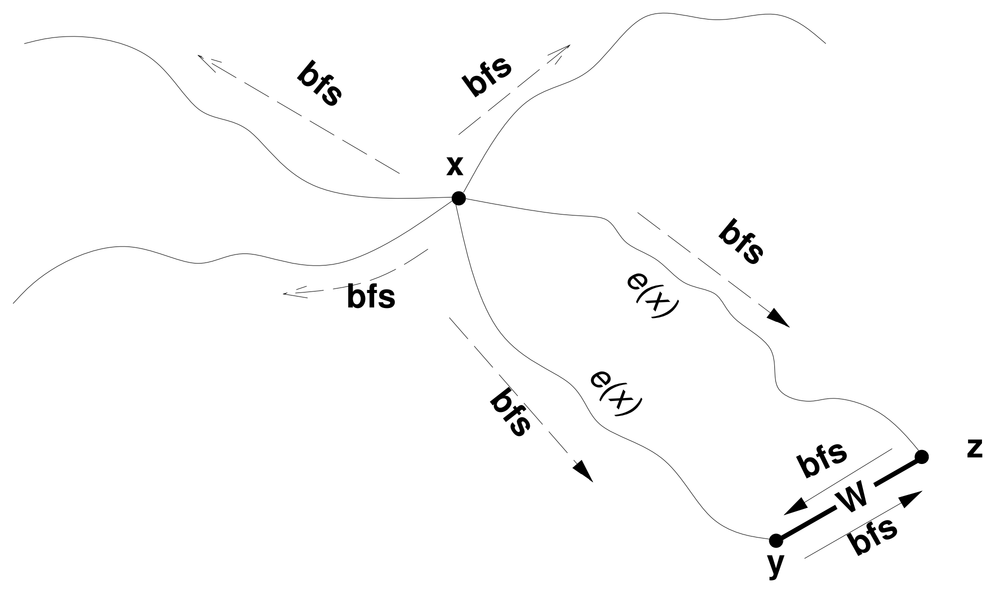

Figure 3.1 : Illustration de l'algorithme **BFS**

vers les autres nœuds. Le deuxième message est alors traité comme spécifié par l'algorithme et conduit donc à l'élimination d'une arête, mais ne produit pas de message supplémentaire.

Si $d(x, y) = w(\mu_1)$ et $w(\mu_1) < w(\mu_2)$ alors le message qui a suivi le chemin μ_1 arrive avant le message qui a suivi le chemin μ_2. Par conséquent, c'est bien le plus court chemin de x à y qui est choisi pour la propagation des messages ***bfs_back***, donc l'arbre construit est un Arbre des Plus Courts Chemins issus de x. $\Box$

Si nous relâchons l'hypothèse de délais parfaitement stables, alors la fonction de poids devient dynamique. Pour être réaliste et plus facilement utilisable, la fonction de poids représente souvent un délai *moyen* de communication. Ainsi l'arbre couvrant construit par l'algorithme sera, suivant les écarts par rapport au délai moyen, *probablement* un arbre des plus courts chemin.

Remarques

-Cette dernière configuration ne correspond pas au modèle $\mathbb{S}$ que nous avons déjà présenté, néanmoins il ne s'agit pas non plus du modèle synchrone $\mathbb{S}'$ défini par rapport aux problème de la tolérance aux pannes. Une extension intéressante de notre travail serait donc de prendre un modèle différent du modèle asynchrone mais pas aussi contraignant que le modèle synchrone ; un modèle où les poids des arêtes seraient des délais moyens de communication et les délais réels seraient des variables aléatoires indépendantes.

-Une autre extension possible du modèle serait de considérer que le poids de l'arc (a, b) peut être différent du poids de l'arc (b, a). En effet, on peut associer aux poids des arcs des valeurs correspondant aux tailles des files d'attente et il n'y a *a priori* aucune raison pour qu'elles soient égales dans les deux sens sur une ligne de communication.
Il s'agit là d'un tout autre problème, que nous ne n'étudierons pas et qui est par ailleurs très peu traité dans la littérature (sauf peut-être par quelques algorithmes de routage comme celui de P. A. Humblet [Humb91]).

Des extensions de la technique de parcours en largeur d'abord pour construire des arbres de plus courts chemins se retrouvent chez de nombreux auteurs tels que [Awer85b, AwGa85, GoHK91, Lavau87, RaHe88], mais aucun d'entre eux ne semble avoir étudié la dernière notion que nous avons présentée.

Une description simple et élégante de l'algorithme n'utilisant pas de façon explicite la variable **attente** peut être réalisée avec la technique de l'appel récursif de procédure distante de G. Florin *et al.* [FlGL93] :

Algorithme **Rec-BFS**

```
Rec-BFS <mesg,orig,f >                (* paramètres passés par adresse *)
   DEBUT-EXMUT ;
   SI père=0 ALORS
      père←orig ; fils←VOISINS −{orig} ;
      FIN-EXMUT ;
      y ←EGO ;
      PAR i SUR fils
         Rec-BFS <mesg,y,f > ON i ;
         DEBUT-EXMUT ;
         SI f=FAUX ALORS fils←fils−{y} ;
         FIN-EXMUT ;
      FIN_PAR
      mon_état←terminé ;
      mesg←bfs_back ;        (* Pour la similitude avec Distri-BFS *)
   SINON
      fils←fils −{y} ;f←FAUX ;
      FIN-EXMUT ;
   FSI
FIN Rec-BFS
```

Cette technique de description pourrait devenir un standard dans l'art de l'écriture des algorithmes distribués lorsque le nombre de messages échangés n'est pas une contrainte forte. En effet tout message envoyé dans la boucle PAR …FIN_PAR doit être associé à un message de «retour de vague» (apportant l'information f=FAUX ou non et le y à ôter éventuellement des fils) pour que le «backtrack» de la récursivité puisse s'opérer normalement.

Les commandes DEBUT-EXMUT et FIN-EXMUT correspondent évidemment respectivement à l'entrée en section d'exclusion mutuelle et sortie de la section en exclusion mutuelle. Ils sont obligatoires dans cet algorithme, car bien qu'il n'y ait qu'une seule vague, la procédure sur un site y peut être appelée en même temps pas deux sites distincts x et z.

Pour calculer le nombre de messages nécessaires à l'exécution de cet algorithme, distinguons les messages de retour de vague par ***ret_vague*** et notons à nouveau a le nombre d'arêtes sur lesquelles deux messages ***bfs*** ont circulé. L'algorithme **Rec-BFS** conduit à l'échange de : $m + a$ messages ***bfs*** comme nous l'avons déjà remarqué auquel il faut ajouter les a messages ***ret_vague*** et les $m - a$ messages ***bfs_back***. Puisque $a = m - n + 1$, il y a en tout $3m - n + 1$ messages échangés.

Notre but, dans ce chapitre, est de décrire les performances de divers algorithmes qui n'ont pas toujours la possibilité de s'écrire simplement sous cette forme sans augmenter de façon importante le nombre de messages, c'est pourquoi nous utilisons plutôt une description de type pseudo-code comme dans la première description.

3.1.2.c De l'arbre couvrant à l'élection

Nous venons de présenter dans le paragraphe précédent la construction d'un arbre couvrant à partir d'un élu. Nous allons présenter maintenant la réciproque.

Après construction d'un arbre couvrant, le processus d'élection dépend des informations déjà disponibles après la construction de l'arbre. Si l'arbre est en fait une arborescence couvrante, et c'est généralement le cas, cela signifie que chaque site connaît son père (et l'ensemble de ses fils, ou plus généralement les lignes de communication qui y conduisent) dans l'arborescence. Par suite l'élection est immédiate (aucun échange de message) car la racine n'a pas de père et tous les autres sites en ont un.

S'il s'agit réellement d'un arbre et non d'une arborescence, la seule information disponible est le nombre de voisins (et par quel lien les contacter) dans l'arbre. Nous en déduisons de suite l'algorithme : les feuilles de l'arbre n'ont qu'un unique voisin, elles envoient un message ***elit*** vers leur père qui collationne ces messages. Quand un nœud «interne» a reçu de tous ses voisins, sauf un, le message ***elit***, il peut envoyer, à son voisin qui est resté muet, ***elit*** avec son identité. A la fin de l'algorithme, deux messages ***elit*** se croisent sur une arête et les receveurs décident localement qui est l'élu par comparaison des identités. Cet algorithme est en fait un algorithme de transformation d'arbre en arborescence. En tout, il faut n messages et $O(h)$ unités de temps où h est la hauteur de l'arbre.

Finalement, nous avons montré que l'élection et la recherche d'un arbre couvrant sont des problèmes équivalents, c'est à dire que l'on peut réduire l'un à l'autre avec une complexité polynomiale, plus précisément en $O(m)$ messages ($m \geq n-1$) et $O(n)$ unités de temps près (si l'on suppose des délais de communication équivalents pour chaque arête). Ce résultat est déjà classique en algorithmique distribué, nous avons seulement exhibé les algorithmes qui permettent d'obtenir cette équivalence et précisé leur complexité ainsi que des extensions possibles.

3.1.3 Autres réductions

Nous allons considérer maintenant d'autres problèmes *réduction*réductibles à la construction d'un Arbre Couvrant : la Recherche d'Extremum et la Terminaison Distribuée.

Considérons le problème de la Recherche d'Extremum. Comme nous l'avons déjà remarqué, ce problème est très proche de celui de l'élection. Précisément, il s'agit, à partir de la même configuration initiale où tous les sites sont dans un état fondamental stable, ayant localement une variable contenant une valeur réelle, d'obtenir la configuration où tous connaissent le maximum (ou le minimum) des valeurs.

Clairement lorsque la recherche d'extremum est réalisée, une élection est réalisée, celui dont la variable contient la même valeur que l'extremum est immédiatement l'élu.

Inversement, si l'on a déjà un arbre couvrant (ou une arborescence couvrante), n (respectivement $n-1$) messages suffisent pour rechercher l'extremum sur l'arbre (voir la précédente réduction), puis $n-1$ messages supplémentaires pour diffuser cet extremum. Globalement nous avons donc l'équivalence Arbre Couvrant, Élection et Recherche d'Extremum. Il est clair que l'on peut étendre ce résultat à tout problème de collecte de variables sur différents sites d'un groupe suivi d'un calcul sur ces variables (dont le problème de la recherche d'extremum n'est qu'un cas particulier).

Considérons maintenant le problème de la Terminaison Distribuée. La configuration initiale du système distribué $\mathbb{S}$st telle que certains sites sont dans l'état terminé, certains autres sont dans l'état travail et des messages sont éventuellement en circulation dans le réseau. La configuration finale est dépendante de la configuration initiale : si il existe au moins un site dans l'état travail alors la terminaison ne peut avoir lieu et la configuration finale est similaire à la configuration initiale, dans le cas contraire, la configuration finale est telle que tous les sites savent que tous les autres sont dans l'état terminé et qu'il n'y a plus de messages en cours de transmission dans le réseau.

Associons la valeur 1 à l'état travail et la valeur 0 à l'état terminé. La Recherche d'Extremum nous donne alors 0 si tous étaient dans l'état terminé. Le fait que plus aucun message n'est en circulation est déjà inclu dans le fait que la recherche d'extremum (au même titre que l'élection) nécessite de parcourir au moins une fois dans chaque sens chaque arête. Comme, d'après nos hypothèses, les messages ne peuvent se doubler sur les lignes de communication, les derniers messages à avoir parcouru le réseau sont donc les messages liés à la recherche d'extremum.

Donc la Terminaison Distribuée est réductible à la construction d'arbre couvrant.

3.1.4 Optimalité ?

Puisque nous sommes amenés à comparer des algorithmes distribués, il est naturel de se poser la question de l'existence d'un « meilleur » algorithme possible pour un problème donné. Un algorithme est meilleur qu'un autre si sa complexité en messages (ou en temps, ou les deux) est plus faible.

Sur le modèle $\mathbb{S}$, que nous avons déjà défini dans le premier chapitre, soit $\mathcal{C}$, la classe des algorithmes résolvant un problème Π sur $\mathbb{S}$. Notons que les algorithmes de $\mathcal{C}$ ne sont pas forcément tous connus.

Nous appelons *complexité* en messages d'un problème Π, la complexité du meilleur algorithme de $\mathcal{C}$ du point de vue de la complexité en message (et de même pour la complexité temporelle). Nous allons étudier ici la complexité de quelques problèmes en rapport avec la construction d'un Arbre Couvrant en milieu distribué.

En séquentiel, seule une mesure de complexité temporelle (et éventuellement une complexité spatiale) est considérée. En algorithmique distribuée, il faut prendre en compte la complexité temporelle et la complexité en nombre de messages échangés. Pour qu'un algorithme de résolution d'un problème Π soit optimal, il faut qu'il atteigne la borne inférieure des complexités des algorithmes de la classe $\mathcal{C}$ du problème Π à la fois en temps et en messages.

Notons (E) le problème de l'élection, (RE) le problème de la Recherche d'Extremum, (AC) le problème de la construction d'un Arbre Couvrant et (ACPM) le problème de la construction d'un Arbre Couvrant de Poids total Minimum.

Théorème 3*Sur un graphe quelconque, (E) est en $\theta(m + n\lg n)$ messages et en $\theta(D)$ unités de temps.*

Preuve : L'article de R G. Gallager *et al.*, entre autres, a prouvé dans [GaHS83] que pour construire un ACPM, il faut $\Omega(m)$ messages et de même pour un AC. En effet, si il existe une arête sur laquelle aucun message n'a circulé, comment être sûr qu'au milieu de cette arête il n'y a pas un nœud ? Donc, il faut au minimum $2m$ messages (chaque arête doit être parcourue dans les deux sens) pour que l'élection soit réalisée ou qu'un arbre couvrant soit construit.

D'autre part, $\Omega(n\lg n)$ messages sont nécessaires pour l'élection sur un anneau (voir l'algorithme de E. J. Chang et R. Roberts dans [ChRo79] et son analyse par C. Lavault [Lavau90a]).

Par conséquent, sur un graphe quelconque la complexité de (E) est $\Omega(m + n\lg n)$. De plus, l'algorithme de R. G. Gallager *et al.* [GaHS83], réalise la construction d'un ACPM avec $O(m + n\lg n)$ messages, ce qui prouve que pour le nombre de messages échangés, (E) est en $\theta(m + n\lg n)$.

Il est évident que (E) est en $\Omega(D)$ en temps ne serait-ce que pour traverser le réseau. L'algorithme de D. Peleg dans [Pele90] a finalement obtenu l'optimalité en temps avec un algorithme en $\theta(D)$ pour l'élection, malheureusement sa complexité en nombre de messages est en $O(mD)$ et n'est donc pas optimale. □

Remarque : La plupart des algorithmes de construction d'arbre couvrant sont en fait, à l'origine, des algorithmes de construction d'ACPM. Ils ont de ce fait une complexité en temps rarement optimale.

L'algorithme de R. G. Gallager *et al.* [GaHS83], par exemple, nécessite $O(m + n\lg n)$ messages donc il est optimal pour le nombre de messages nécessaires mais sa complexité en temps est en $O(n\lg n)$ dans le pire des cas. La complexité temporelle a été améliorée par B. Awerbuch dans l'algorithme présenté dans [Awer87] qui nécessite $\theta(n)$ unités de temps et qui peut être alors optimal pour le problème de la construction d'un ACPM sur des graphes où $D = O(n)$. Malheureusement, comme l'a montré B. Bollobás dans [Boll84], le diamètre d'un graphe est « souvent » (graphes aléatoires réguliers ou graphes aléatoires de degré suffisamment grand) en $O(\lg n)$; par conséquent, l'algorithme de B. Awerbuch n'est pas optimal en temps.

Finalement, le problème de l'existence d'un algorithme obtenant l'optimalité à la fois en temps et en messages, est un problème ouvert. On retrouve là la problématique classique sur le compromis espace temps en informatique.

Corollaire 1*Sur un graphe quelconque, (AC) et (RE) sont en $\theta(m + n\lg n)$ messages et en $\theta(D)$ unités de temps.*

Preuve : Immédiat d'après le théorème précédent et l'équivalence entre (E), (AC) et (RE). □

Théorème 4 *Sur un graphe quelconque, (ACPM) est en $\theta(m + n\lg n)$ messages et en $\Omega(D)$ et $O(n)$ unités de temps.*

Preuve : D'après l'algorithme de B. Awerbuch [Awer87], (ACPM) est en $\theta(m + n\lg n)$ messages et $O(n)$ unités de temps. Puisque (AC) est en $\theta(D)$, alors (ACPM) est en $\Omega(D)$. □

Pour des raisons de simplicité, nous exposons et codons l'algorithme de [GaHS83] et pas celui de [Awer87], théoriquement meilleur du point de vue du temps mais il n'apporte que cet aspect théorique. Ainsi, l'algorithme de [Awer87] n'est pas primordial dans le cadre de la partie pratique de notre étude qui est consacrée aux messages et aux problèmes associés. De plus, nous avons par ailleurs déjà exprimé le caractère paradoxal de la notion de temps en algorithmique distribuée asynchrone (voir paragraphe 1.5.2).

Nous verrons que d'autres algorithmes complètent la construction d'un Arbre Couvrant en $\theta(m + n\lg n)$ messages. Toutefois, les résultats obtenus en pratique, que nous verrons au chapitre 4, permettent de les départager suivant différents critères tels le degré du réseau la taille mémoire, etc.

3.1.5 Problème de l'Arbre Couvrant

Nous avons déjà défini dans le premier chapitre ce qu'était un arbre couvrant, le problème de sa construction sur un graphe quelconque est un problème classique de la théorie des graphes même si historiquement il est postérieur au problème de la recherche d'Arbre Couvrant de Poids total Minimum dont on trouve des exemples d'algorithmes dès le début du siècle[1].

La recherche d'un ACPM, avec des poids statiques, n'est pas très réaliste si l'on considère ces poids comme étant des délais de communication dans un réseau. Ces poids, pour avoir quelque intérêt dans la réalité, devraient illustrer le coût de transfert de message pour chaque arête. Or, ce coût est dynamique puisqu'il résulte de la charge du réseau. Sur des réseaux dynamiques, il vaut mieux construire des tables de routages dynamiques multi-chemins en chaque nœud plutôt qu'un Arbre Couvrant de Poids total Minimum, ainsi des techniques de routage spécifiques peuvent être mises en œuvre telles que la détection de

[1] Otakar Borůvka, 1926. Son algorithme est détaillé dans l'historique du problème de l'ACPM de R. L. Graham et P. Hell [GrHe85]

surcharge d'une ligne qui entraîne un détournement partiel vers une autre ligne moins chargée. De telles techniques sont applicables à la structure d'arbre en elle-même mais la contrainte de poids total minimum ne correspond pas à la recherche d'optimalité dans les réseaux de communication sauf si les poids représentent le coût « au mètre » de la ligne de communication, ainsi baisser le coût total suit alors l'intérêt économique. De même si l'opération de diffusion de messages sur tout le réseau est une opération de base à effectuer souvent, l'ACPM est la structure de contrôle donnant le meilleur coût.

Les algorithmes de construction d'AC que nous allons étudier dans ce chapitre comportent un certain nombre de caractéristiques similaires : la construction se fait par fusion de fragment et tout fragment (ou domaine) est un arbre disposant d'une identité et d'une racine. Au départ, tout site initiateur est un fragment à lui tout-seul, avec sa propre identité en guise d'identité de fragment. Certains algorithmes utilisent la notion de phase pour limiter la complexité en nombre de messages échangés. Cette notion engendre des phénomènes d'attente, plus ou moins marqués suivant les algorithmes et surtout induit une certaine perte de parallélisme voire une forme de séquentialisation pure et simple. La notion de phase consiste en la suppression de deux fragments de même phase (ou jetons de même phase suivant l'algorithme considéré) pour n'en former qu'un seul de phase supérieure, ce qui implique que le nombre de phases est alors au plus égal à $\lceil \lg n \rceil$.

Nous allons étudier dans ce chapitre les algorithmes de construction d'arbre couvrant de [Bute93, LaLa89a, KoKM90], seul le dernier utilise la notion de phase, mais cette technique se retrouve aussi dans la construction d'Arbre Couvrant de Poids Minimum.

3.1.6 Problème de l'Arbre Couvrant de Poids Minimum

Les algorithmes de construction d'Arbre Couvrant de Poids Minimum (que l'on note ACPM) utilisent les propriétés suivantes :

Propriété 1 (R. C. Prim 1957, G. Sollin 1961)*Pour tout sommet x d'un graphe, soit $e = (x, y)$ l'arête adjacente à x de poids minimum. Il existe un arbre couvrant de poids minimum contenant l'arête e [Prim57, Soll61].*

Propriété 2 (R. G. Gallager et al. **1983)***Étant donné un fragment f d'un ACPM, soit e une arête sortante du fragment telle qu'elle soit de poids minimum. L'ajout de e (et de son nœud adjacent) au fragment f forme un autre fragment d'un ACPM [GaHS83].*

Propriété 3*Si toutes les arêtes d'un graphe connexe ont des poids différents alors l'ACPM est unique.*

Les propriétés locales telles que la propriété 1, sont essentielles car elles renvoient aux fondements même de l'algorithmique distribuée.

Les propriétés 2 et 3, déduites de la première, permettent d'imaginer une méthode de construction générale des ACPM à partir d'un graphe où les poids des arêtes sont tous différents. En effet, la propriété 2 permet de faire croître chaque fragment dans n'importe quel ordre. Quand deux fragments d'ACPM ont un nœud en commun, la propriété 3 assure que l'union de ces deux fragments est toujours un fragment de l'ACPM.

En séquentiel, les algorithmes de R. C. Prim, E. W. Dijkstra [Prim57, Dijk59] commencent par un simple nœud considéré comme un fragment auquel on ajoute successivement arêtes et nœuds jusqu'à ce que le fragment recouvre l'ensemble des nœuds du graphe. Cette technique a donc recours aux propriétés 1 et 2.

L'algorithme de J. B. Kruskal [Krus56] commence par trier les arêtes par ordre de poids croissant, tous les nœuds étant considérés initialement comme des fragments. Les arêtes sont ajoutées successivement à l'ensemble des fragments si et seulement si elles ne créent pas de circuit.

L'algorithme de G. Sollin [Soll61] est le plus proche des algorithmes distribués : à chaque étape, il faut joindre un sommet quelconque à son plus proche voisin (au sens du poids de l'arête les reliant) de sorte que l'on ne produit pas de cycle. Si le graphe obtenu est connexe, l'algorithme est terminé, sinon les composantes connexes obtenues sont considérées à leur tour comme des sommets et l'algorithme est réitéré.

A. C.-C. Yao et d'autres auteurs utilisent une approche similaire à l'approche distribuée : au départ tous les nœuds sont des fragments, chaque fragment est étendu par fusion avec d'autres jusqu'à l'obtention d'un ACPM. L'algorithme de A. C.-C. Yao [Yao75] oblige de plus une croissance équilibrée des fragments.

Les algorithmes de J. B. Kruskal, R. C. Prim, E. W. Dijkstra et G. Sollin peuvent s'exécuter si tous les poids des arêtes ne sont pas tous distincts à la condition de définir un ordre arbitraire. Par contre, celui de A. C.-C. Yao ne peut, sans de profondes modifications, accepter le relâchement de cette hypothèse.

En distribué, les algorithmes utilisent les mêmes principes. Les difficultés liés au parallélisme sont surtout liés à la résolution d'éventuels conflits entre fragments. Ces conflits sont arbitrés par les poids des arêtes lorsqu'elles sont distinctes ou par les identités des nœuds du graphe (qui sont supposés distincts).

Tous les algorithmes procèdent par fusion (concaténation) de fragments jusqu'à ce qu'il n'y ait plus qu'un seul fragment contenant tous les sommets. Ce fragment restant est alors l'ACPM. Nous allons étudier dans ce chapitre les algorithmes de [GaHS83, LaRo86]. En distribué, obtenir un ordre arbitraire sur les poids des arêtes est coûteux, même si l'on dispose de l'hypothèse d'identités toutes distinctes, il faut, dans le pire des cas et sans modification profonde de l'algorithme considéré, $O(m)$ messages supplémentaires pour obtenir cette distinction.

3.1.7 Problème de l'Arbre Couvrant de Diamètre Minimal

Parmi les arbres couvrants d'un graphe, il est intéressant d'examiner ceux de diamètre minimal. En effet, la propriété de diamètre minimal apporte à la structure d'arbre une minimisation des temps de transfert des messages d'un bout à l'autre de cette structure.

Nous n'avons pu trouver d'algorithme distribué de construction, même sur des graphes réguliers. Il existe pourtant un problème, en apparence similaire, qui s'appelle l'Arbre Couvrant Géométrique de Diamètre Minimum qui consiste à construire un arbre couvrant de diamètre minimum sur un ensemble de points du plan Euclidien. Ce problème a été résolu par J.-M. Ho *et al.* ainsi que par E. Ihler *et al.* [HLCW91, IhRW91] avec des algorithmes séquentiels polynomiaux. Nous n'exposons pas ici les méthodes de résolution utilisées qui s'avèrent être assez éloignées de celles que nous allons utiliser pour la recherche d'un ACDM, qui plus est, elles sont essentiellement séquentielles. Notre méthode de résolution est présentée en deux parties : la résolution d'un cas particulier puis le cas général. Cette méthode donne à la fois un algorithme séquentiel et un algorithme distribué.

3.1.7.a Le problème

Avec les notations du paragraphe 1.2, le problème se décrit comme suit : parmi l'ensemble des arbres couvrants de $G = (X, U)$, graphe non-orienté, valué positivement et connexe ; trouver un arbre T tel que son diamètre ($D(T)$) soit minimum.

Le problème réduit au plan Euclidien (ACGDM) a été résolu par [HLCW91, IhRW91] avec des algorithmes séquentiels en $O(n^3)$. Le problème de la construction d'un ACDM est une généralisation de l'ACGDM, car on peut construire un graphe (complet) avec l'ensemble des points reliés deux à deux par des arêtes de poids égal à la distance Euclidienne.

Indépendamment de nos travaux, nous avons trouvé dans l'article de P. M. Camerini *et al* [CaGM80] la preuve que ce problème, en séquentiel, est polynomial.

Cet article n'est parvenu à notre connaissance que très tardivement car ces auteurs utilisent une autre appellation pour ce problème : le *max-path*, d'ailleurs cet article à été ignoré par [HLCW91, IhRW91]. La démonstration qu'ils apportent consiste classiquement en la réduction polynomiale de ce problème à un autre problème qui à déjà été prouvé comme étant polynomial : le problème de la recherche de centre absolu (voir par exemple le livre de N. Christophides : [Chri75]). La fusion d'une des méthodes de N. Christophides et de la réduction polynomiale permet d'obtenir un algorithme qui, bien que d'une approche différente, est finalement assez proche de celui que nous allons présenter.

3.2 Algorithmes de construction d'Arbres Couvrants

3.2.1 Principes généraux des algorithmes étudiés

Tous les algorithmes de recherche d'AC ou d'ACPM procèdent par fusion (concaténation) de fragments jusqu'à ce qu'il n'y ait plus qu'un seul fragment contenant tous les sommets. Ce fragment restant est alors l'AC (ou ACPM suivant les cas).

Ces algorithmes diffèrent par la façon dont est gérée la fusion entre fragments. Dans les algorithmes de I. Lavallée, C. Lavault et G. Roucairol [LaLa89a, LaRo86], un fragment est une arborescence couvrante (logique) dont le site privilégié est la racine de cet arborescence. L'identité du fragment est celle de la racine. Dans ces deux algorithmes l'identité finale de la racine est complètement déterminée avant même le début de l'algorithme : c'est le nœud d'identité minimale (ou maximale suivant le sens choisi pour la relation d'ordre sur les identités des sites). Dans [Bute93], les fragments sont aussi des arbres mais l'identité d'un fragment peut être distincte de l'identité de la racine du fragment, ainsi, à la fin de l'algorithme, l'élu (la racine du dernier fragment) est quelconque voir la figure 3.2.

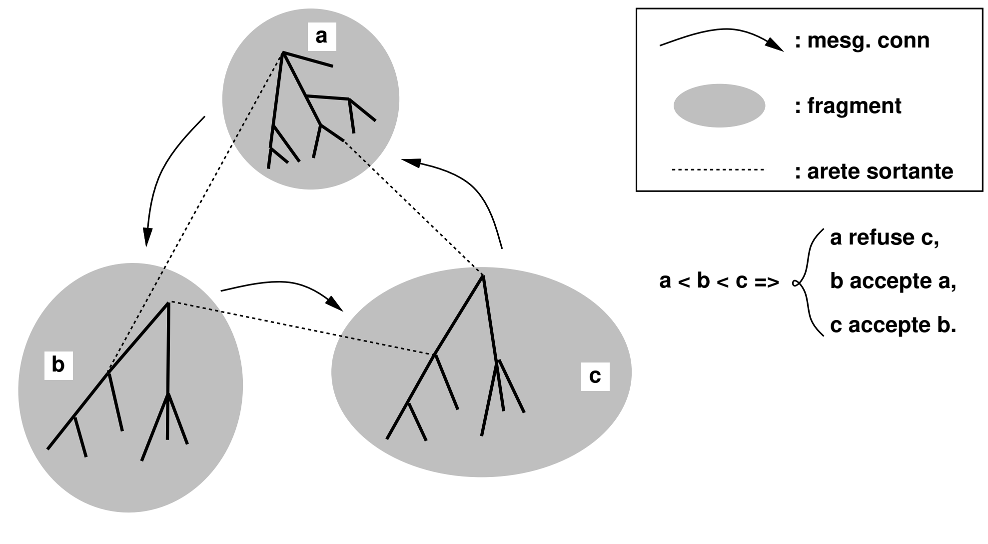

Figure 3.2 : Construction d'AC : exemple de fusions de fragments

Dans l'algorithme de [KoKM90], les fragments sont des domaines ayant une identité et une phase, ils sont caractérisés par des jetons qui tentent d'annexer de nouveaux sites au domaine. La fusion de fragments est alors la rencontre de deux jetons, détruits pour la construction d'un nouveau de phase plus élevée et dont l'identité dépend du site où s'est produit la rencontre. En conséquence, l'élu (la racine de l'arbre couvrant) est aussi quelconque.

3.2.2 Algorithme avec racine pré-déterminée

Nous allons expliciter ici l'algorithme de I. Lavallée et C. Lavault décrit dans [LaLa89a], qui est une amélioration notoire de l'algorithme de I. Lavallée et G. Roucairol [LaRo86] en ce qui concerne le nombre et la taille des messages échangés.

Dans cet algorithme, les fragments fusionnent de la façon suivante: si un fragment d'identité i demande à se fusionner avec un fragment d'identité j tel que $i < j$, alors la fusion est acceptée (si la relation d'ordre n'est pas vérifiée, alors la demande est rejetée). Le nouveau fragment, produit par fusion, a comme identité j et il contient le sous-arbre obtenu par concaténation du sous-arbre du fragment i avec celui du fragment j en utilisant pour la concaténation l'arête qui a servi à transmettre la demande de fusion entre les deux fragments. Par construction, cette arête est unique, voir [LaLa89a].

Cette méthode a déjà été utilisée dans [LaRo86], elle est intéressante à plus d'un titre : parallélisme des fusions ainsi que multiplicité. En effet, plusieurs fragments peuvent se fusionner à un autre, et un fragment peut fusionner avec un autre alors que cet autre est déjà en train de ce fusionner avec un troisième etc... Il est donc particulièrement difficile d'analyser cet algorithme précisément, surtout si l'on s'intéresse à l'analyse de la complexité en moyenne !

Pour pallier à certains défauts de l'algorithme [LaRo86] (longueur des messages en $O(n)$ et nombre total de messages non borné dans le pire des cas), en [LaLa89a], il a été introduit en chaque site une gestion en chaque site des arêtes adjacentes à ce site par l'utilisation de tableaux $Actif[]$ et $Candidat[]$.

Nous avons relevé une léger défaut dans l'algorithme qui peut dans certains cas particuliers, conduire à une terminaison par message au lieu d'une terminaison par processus. Nous apportons donc une correction possible qui permet de retrouver la terminaison souhaitée.

Nous décrirons ici le fonctionnement de l'algorithme par l'intermédiaire de ses réactions au vu des événements (messages...) rencontrés au cours de l'exécution de l'algorithme. Le codage proprement dit de l'algorithme est donné en annexe.

3.2.2.a Messages échangés

Les messages échangés sont de type $<$ *identité*, *mot_clé*, *booléen*, *booléen* $>$, où *identité* est l'identité de la composante émettant le message (maximum des identités des processeurs de la composante) ;

mot_clé précise la nature du message (il caractérise l'événement). C'est un élément de l'ensemble {***conn***, ***ok***, ***nok***, ***cousin***, ***nrac***, ***maj***, ***ouvre***, ***fin***} où :

- ***conn*** désigne une tentative de connexion,
- ***ok*** représente une réponse affirmative à la tentative précédente,
- ***nok*** représente une réponse négative,
- ***cousin*** représente une réponse négative, précisant que le nœud appartient au même fragment,
- ***nrac*** est utilisé lors de la mise à jour de l'identité d'un fragment,
- les messages ***maj*** et ***ouvre*** entraînent la mise à jour des $Actif[\,]$ et $Candidat[\,]$,
- Le message ***fin*** est émis pour diffuser la terminaison de l'algorithme.

3.2.2.b Variables

Tout site gère son père et ses fils dans son fragment ainsi que l'identité de la racine du fragment : *root* (qui donne aussi l'identité du fragment) et l'état des arêtes menant vers ses voisins : $Actif[\,]$ et $Candidat[\,]$.

3.2.2.c Gestion des arêtes adjacentes

On définit pour tout processus x et pour tout voisin y de x, les propriétés suivantes :

- y est *candidat* pour x si et seulement si y appartient *a priori* a un autre fragment d'identité supérieure à celle de x ou bien si y est un fils de x et y est ouvert.
- x est *ouvert* si et seulement s'il existe au moins un voisin qui est candidat pour x.

Informellement, si y est candidat pour x c'est qu'il est possible d'étendre le fragment auquel appartient x via une tentative de connexion sur y qui lui même peut soit appartenir effectivement à un autre fragment ou bien être un fils de x et dans ce cas propager la tentative de connexion.

•y est *actif* pour x si et seulement si y appartient *a priori* à un autre fragment ou bien si y est un fils de x et y est libre.

•x est *libre* si et seulement si il existe au moins un voisin qui est actif pour x.

Informellement, le concept actif est nécessaire à la terminaison par processus de l'algorithme. En effet, si un site a au moins un voisin actif alors il est possible que ce voisin appartienne à un autre fragment – par conséquent l'algorithme n'est pas terminé puisque l'on veut construire un arbre couvrant. Le concept actif est répercuté de fils en père, comme la propriété de candidat, via une fonction de type **ou** logique (si au moins un des fils de x est candidat pour x alors le site x est candidat pour son père et ainsi de suite).

Aux propriétés «être actif pour» et «être candidat pour» sont associées respectivement deux tableaux de booléens : $Actif[\,]$ et $Candidat[\,]$. De même, aux propriétés «être libre» et «être ouvert» sont associées respectivement deux variables booléennes : libre et ouvert.

Remarque : Si on note Var_y la variable Var du processus y (voisin de x) telle que l'on a pu la déduire d'après les messages échangés, il est important de remarquer que $Var_y = \alpha$ ne signifie pas $Var = \alpha$ au même instant pour le processus y. En effet, notre hypothèse d'asynchronisme implique qu'il n'y a aucun moyen, dans le cas général, de s'assurer qu'une variable ait effectivement une valeur donnée à un instant donné. Donc lorsque l'on écrit y est actif pour x, cela ne signifie pas forcément que y à cet instant soit réellement dans un autre fragment.

Si x ne dispose d'aucune information sur un voisin y (c'est à dire qu'aucun message n'a été émis de y vers x) x le considère par défaut comme actif et candidat.

La variable attente_réponse, du processus x, contient la valeur VRAI si et seulement si un message ***conn*** a été émis de x vers un voisin y (cette variable n'a donc de sens que pour la racine du fragment qui seule possède le privilège d'envoyer ces messages) et que la réponse n'a pas encore été reçue. Il faut remarquer que le fait d'attendre une réponse à une demande de connexion n'empêche pas d'accepter (ou de refuser) des demandes d'autres voisins (y compris de y car deux messages peuvent se croiser sur une ligne).

3.2.2.d Démarrage de l'algorithme

Au départ, pour tout site éveillé, l'ensemble de ses voisins est à la fois actif et candidat, il est donc ouvert et libre. Il n'a à cet instant ni père ni fils et l'identité de son fragment est égale à son identité, il est alors racine de son propre fragment.

Pour lancer l'algorithme, un ou plusieurs nœuds s'éveillent spontanément (pas forcément en même temps) et envoient à un de leurs voisins un message ***conn*** exprimant une demande de fusion.

Un nœud non éveillé qui reçoit un message s'éveille alors immédiatement, et peut, de façon non déterministe[2], soit envoyer un message ***conn***, soit lire le message reçu. Il est possible d'améliorer l'algorithme de façon assez significative si on considère qu'en cas de réveil par message, un nœud se «fusionne» spontanément à l'émetteur.

3.2.2.e Schéma de principe de l'algorithme

La procédure **TENTE_CONNEXION** consiste en un choix aléatoire d'un voisin candidat vers lequel l'algorithme peut envoyer un message <*a*,***conn***>.

L'algorithme s'articule de la façon suivante :

[2]Il y a ici un certain abus de langage, ce n'est pas le non déterminisme au sens mathématique du terme comme peut l'être la machine de Turing non déterministe, mais plutôt un choix aléatoire, pas forcément équiprobable, que l'on peut qualifier de non prédéterminé par le réseau.

Algorithme **Distri-AC1**

INIT;

TANT QUE mon_état $\neq$ `terminé` FAIRE A, B ou C : *(* Choix aléatoire *)*

$\boxed{\text{A}}$: SI racine=EGO ET attente_reponse=FAUX ET ouvert ALORS

  TENTE_CONNEXION ;

$\boxed{\text{B}}$: SI racine=EGO ET attente_reponse=FAUX ET libre=FAUX ALORS

  POUR TOUT $i \in fils$ ENVOIE à i $<\boldsymbol{fin}>$;

  mon_état$\leftarrow$ `terminé` ;

$\boxed{\text{C}}$:

  Soit EGO reçoit de y un message $< a,\boldsymbol{conn}>$ ALORS

    ...

  Soit EGO reçoit de y un message $< a,\boldsymbol{ok}>$ ALORS

    ...

  Soit EGO reçoit de y un message $<\boldsymbol{nok}>$ ALORS

    ...

  Soit EGO reçoit de y un message $<\boldsymbol{cousin}>$ ALORS

    ...

  Soit EGO reçoit de y un message $<\boldsymbol{ouvre}>$ ALORS

    ...

  Soit EGO reçoit de y un message $< a,\boldsymbol{nrac}>$ ALORS

    ...

  Soit EGO reçoit de y un message $<\boldsymbol{maj}, c, d >$ ALORS

    ...

FIN TANT QUE.

3.2.2.f Règles de fonctionnement

- Seule la racine du fragment peut envoyer un message ***conn*** vers un candidat (la condition « est racine du fragment » est testé par racine=EGO).
- Lorsqu'un nœud a émis un message ***conn***, il attend la réponse avant d'en envoyer un autre (variable *attente_reponse*). La tentative de connexion est propagée jusqu'à une feuille du fragment comme le montre la figure 3.3 (les nœuds en grisé sont les racine des fragments.
- Un nœud accepte un message ***conn*** si et seulement si l'identité du fragment porté par le message est strictement inférieure à la sienne.
- Après réception d'un message, les tableaux $Actif[\,]$ et $Candidats[\,]$ sont mis à jour conformément aux propriétés énoncées précédemment.

•Si pour une racine, il n'y a plus de voisin (y compris les fils) actif, l'algorithme s'arrête (cas $\boxed{\text{B}}$ de l'algorithme).

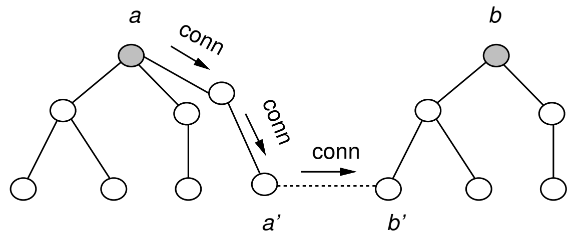

Figure 3.3 : Tentative de connexion dans [LaLa89a]

Nous allons décrire maintenant la réaction qu'adopte le site EGO lorsqu'il reçoit un message d'un site y. Nous avons relevé une légère erreur pour la bonne terminaison de l'algorithme, nous la justifions et proposons une correction.

3.2.2.g Réception d'un message *conn*

Trois cas se présentent :

1. Soit l'identité du fragment de y est inférieure à l'identité du fragment auquel appartient EGO et alors il y a fusion. Pour réaliser cette fusion, EGO répond par un message ***ok*** et désormais le considère comme un fils, par défaut non-candidat.

2. Soit les identités des fragments sont identiques, alors les deux nœuds appartiennent au même fragment. Dès lors, EGO ne considère plus ce voisin comme actif et par conséquent plus comme candidat. De plus, il lui renvoie un message ***cousin***.

   Remarque : Un cas rare peut alors survenir : imaginons que l'arête (EGO, y) soit la dernière à être explorée. Cela signifie que EGO était libre avant de recevoir ce message, car il ne savait pas encore qu'il s'agissait d'un cousin. Après ce message, puisque (EGO, y) était la dernière arête non éliminée, EGO n'est plus libre, pourtant son père le considère toujours comme avant, c'est à dire actif. Donc le père de son père et ainsi de suite la racine du fragment a au moins un fils actif, soit x. Considérons maintenant y, recevant le message ***cousin*** de y, il peut constater qu'il n'a plus d'arête sur laquelle tenter une connexion, donc, il envoie un message ***nok*** qui sera propagé jusqu'à la racine. Lorsque la racine reçoit ce message, supposons, ce qui est probable, qu'il provienne d'un fils différent de x.

La racine du fragment a donc encore au moins un fils actif : x. Malheureusement plus aucun message ne sera émis...
Par conséquent l'algorithme termine seulement par message dans ce cas. Pour lever ce défaut, il faut détecter ici si EGO devient fermé après la réception du message ***conn***, et si c'est le cas envoyer un message de mise à jour (***maj*** par exemple) vers son père.

3. Soit l'identité du fragment de y est supérieure, alors la connexion est rejetée. EGO renvoie alors un message ***nok*** et considère ce voisin comme candidat (même s'il ne l'était plus). Il ne peut pas le considérer d'emblée comme père car cela reviendrait à autoriser des demandes de connexion par d'autres nœuds que la racine du fragment et donc contredirait les règles précédentes.

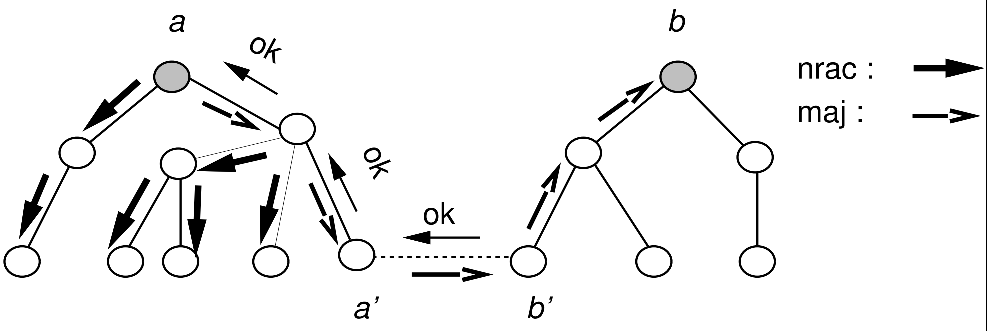

Figure 3.4 : Le message ***ok*** dans [LaLa89a]

3.2.2.h Réception d'un message *ok*

La figure 3.4) illustre les deux situations suivantes :

- Soit ce message provient d'un fils y et alors le receveur doit effectuer les actions suivantes :
  - Propager le message ***ok*** vers son père,
  - Considérer son père comme un fils,
  - Affecter à la variable père l'identité y.
  - mettre à jour l'identité de fragment des sites du sous-arbre enraciné en EGO via une diffusion de message ***nrac***.

C'est à dire que, suite à la remontée du message ***ok*** vers la racine du fragment absorbé, le sens fils-père sur cette branche est inversé (ceci est la méthode de retournement de chemin qui a été utilisée par la suite par M. Naïmi, M. Tréhel et A. Arnold dans [TrNa87, NaTA88] sous le nom de « Path reversal »).

- Soit ce message provient d'un voisin non-fils et il a alors transité par une arête dont chaque extrémité appartient à deux fragments différents. y est considéré alors comme le père de EGO. De même que dans le premier cas, il doit propager le message vers son ancien père qu'il considère désormais comme un fils.

Si EGO n'avait pas de père, alors il était racine du fragment et doit maintenant diffuser la nouvelle identité de fragment vers ses fils par le message ***nrac***. Il envoie ensuite un message ***maj*** vers son nouveau père (y).

Dans les cas précédents, il y a mise à jour des tableaux $Actif[\,]$ et $Candidat[\,]$ puisqu'un père n'est ni actif ni candidat. Le nouveau fils est, quant à lui, considéré comme seulement actif (jusqu'à l'éventuelle modification apportée par ***maj***).

Enfin, EGO prend pour identité de fragment celle que le message véhicule.

3.2.2.i Réception d'un message *nok*

Nous devons distinguer deux cas :

- Soit le message provient d'un fils, et si EGO n'a pas de voisin (ou fils) candidat, alors le message doit être répercuté vers son père. Dans le cas où EGO a au moins un voisin candidat, il peut renvoyer un message ***conn*** vers ce candidat.
- Soit le message provient d'un voisin non-fils et par suite EGO ne doit plus considérer y comme candidat. Ensuite, EGO se comporte comme dans le cas précédent.

Si EGO n'a pas de père, c'est qu'il est racine du fragment et donc qu'il peut alors affecter attente_réponse à FAUX et ainsi s'autoriser à émettre un nouveau message ***conn*** vers un candidat. Dans le cas particulier où il n'a plus de candidat, mais qu'il est encore *libre*, il ne peut qu'attendre un message de type ***maj*** pour changer d'état.

Dans les cas précédents, la mise à jour des tableaux $Candidat[\,]$ et $actif[\,]$ suit la remontée vers la racine du message ***nok***.

3.2.2.j Réception d'un message *cousin* de y

Ce message est une réponse négative à une tentative de connexion, il signifie que le voisin y appartient au même fragment que EGO.

La réaction est alors la suivante : suppresion de y des actifs et candidats ; puis, si cela est possible, tenter une nouvelle connexion, sinon un message ***nok*** est émis vers le père de EGO.

Si EGO n'a pas de père, comme précédemment, c'est qu'il est racine du fragment et ainsi qu'il peut affecter attente_réponse à FAUX.

3.2.2.k Réception d'un message *maj*

Ce message conclut l'opération de fusion ; il est équivalent à une mise à jour inconditionnelle.

Le message ***maj*** ne peut provenir que d'un fils et apporte comme information l'état des variables libre et ouvert de l'émetteur. EGO doit donc prendre en compte ces valeurs pour modifier en conséquence ses tableaux $Actif[\,]$ et $Candidat[\,]$. Ces modifications entraînent éventuellement une évolution de ses variables libre et ouvert. Les (nouvelles) valeurs de ces variables sont enfin transmises par un message ***maj*** vers son père, pour propager la mise à jour.

3.2.2.l Réception d'un message *nrac*

Ce message ne peut venir que du père de EGO, il est destiné à la mise à jour de l'identité du fragment et est diffusé sur l'ensemble des nœuds de celui-ci. Par conséquent EGO prend la nouvelle identité de fragment et diffuse le message ***nrac*** à ses fils.

3.2.2.m Réception d'un message *fin*

Ce message est un message de terminaison. Il est émis par la racine (du dernier fragment restant) lorsqu'elle n'a plus de voisins (y compris ses fils) actif, c'est à dire que la variable libre a été affectée à la valeur FAUX. Par conséquent, ce message peut être la conséquence de tout message susceptible de modifier le tableau $Actif[\,]$.

Tout receveur d'un message ***fin*** doit d'abord le propager vers ses fils puis s'arrêter.

3.2.2.n Complexité et analyse

Nous ne détaillons pas complètement l'analyse de cet algorithme ici. Cette analyse est du même type que celle de l'algorithme suivant et cette dernière possède l'avantage d'être plus simple car l'algorithme a d'une part moins de mots-clés et d'autre part ces mots clés sont différents suivant leur lieu de circulation (hors fragment ou non).

Distinguons donc les messages de type ***nok*** émis entre les fragments et ceux émis à l'intérieur des fragments que l'on note ***noki***. De même, nous faisons cette distinction pour ***conn*** et ***conni***, ***ok*** et ***oki***.

Les relations suivantes sont alors vérifiées :

$$
\begin{aligned}
N_{fin} &= n-1; && (3.1)\\
N_{ok} &= n-1; && (3.2)\\
N_{cousin} &= m-(n-1); && (3.3)\\
N_{conn} &= N_{ok}+N_{nok}+N_{cousin}\,; && (3.4)\\
N_{maj} &\geq N_{ok}+N_{oki}\,; && (3.5)
\end{aligned}
$$

Pour les équations (3.1), (3.2) et (3.3) ; $n-1$ est le nombre d'arêtes de l'Arbre Couvrant trouvé par l'algorithme. Plus précisément, l'équation (3.3) provient du fait qu'une arête sur laquelle un message ***cousin*** a circulé n'est plus considérée dans le reste de l'algorithme et est exclue pour la construction de l'Arbre Couvrant (dans le même esprit que l'algorithme **BFS** en début de chapitre).

L'equation 3.5 provient directement de la technique de construction : tout message ***ok*** reçu est propagé jusqu'à la racine du fragment (par des messages ***oki***) qui renvoie alors un message ***maj*** suivant le chemin pris par les messages ***oki***. D'autre part, le message ***maj*** doit remonter dans l'autre fragment jusqu'à la racine pour compléter la fusion.

Les messages internes ***conni*** et ***noki*** représentent en quelque sorte un parcours de graphe qui est ni en largeur d'abord, ni en profondeur d'abord. Les messages ***noki*** représentent les « retour-arrière ». Le parcours est stoppé dès que l'on trouve une arête sortante du fragment telle qu'une tentative de connexion via cette arête reçoit une réponse ***ok***. Il coûtent donc de l'ordre de $2m$ messages.

Les messages de mise à jour : ***maj*** et ***nrac*** peuvent conduire à échanger $O(n^2)$ messages dans certains cas : prenons par exemple un anneau dont les sites sont rangés dans l'ordre des identités croissantes. Pour simplifier, numérotons les de 1 à n. Supposons que tous les sites s'éveillent spontanément et qu'ils tentent de se connecter vers le plus grand de leurs voisins.

Nous allons dérouler le fonctionnement de l'algorithme « phase après phase », c'est à dire en faisant l'hypothèse que le réseau est à peu près synchrone.

1. A la tentative de connexion de n, la réponse est ***nok***, dans les autres cas la réponse est ***ok***.
2. Pour chaque message ***ok***, un message ***maj*** est renvoyé. n tente alors de faire une connexion vers 1.
3. Chaque message ***maj*** est alors renvoyé vers le père sauf celui reçu par n.

etc.

En tout, il circule dans ce cas n messages ***conn***, $+ \; n-1$ messages ***ok***, + 1 message ***ok***, $+ \sum_{i=1}^{n-1} i$ messages ***maj***. Nous n'avons pas illustré le comportement du message ***nrac*** mais il est clair que sur cet exemple il aura aussi une complexité en $O(n^2)$.

Dès lors, l'algorithme de [LaLa89a] a une complexité en $O(n^2)$ messages dans le pire des cas. La complexité temporelle est estimée à $O(n \lg n)$.

Nous voyons clairement que c'est sur le procédé de mise à jour de l'identité d'un fragment (que ce soit via le message ***maj*** ou le message ***nrac***) que vont se distinguer les algorithmes distribués de construction d'Arbre Couvrant. Le diamètre du réseau joue alors un rôle important : plus il est grand plus la profondeur des fragments est grande et ainsi plus coûteuses sont les mises à jour. Par conséquent cet algorithme a un comportement intéressant dès que le réseau est de faible diamètre donc par exemple dès lors que le graphe des communications est dense.

La gestion par phase permet de ne faire ces mises à jour que *de temps en temps* c'est à dire uniquement à chaque changement de phase. Ainsi, on pourrait imaginer une transformation de cet algorithme avec une gestion par phase comme celle qu'utilise R. G. Gallager *et al.* [GaHS83] et que nous allons détailler dans l'étude de la construction d'AC de poids total minimum. Nous verrons que cette gestion introduit des temps d'attente mais évite l'apparition de ces « cas pathologiques ». Ils sont appelés ainsi au vu de la nécessité d'un ordre particulier sur les réceptions de messages, par conséquent d'occurrence rare sur des réseaux réellement asynchrones.

3.2.3 Algorithme à élection non pré-déterminée

Dans cet algorithme [Bute93], en plus de l'obtention d'un Arbre Couvrant, il y a élection d'un processus dont on ne peut, *a priori*, prévoir l'identité. Il s'agit là d'une étape importante dans la recherche d'un algorithme élaborant une structure de contrôle réellement résistante aux pannes. En effet, si l'élu est prévisible, il peut être la cible d'actions extérieures et, si c'est toujours le même, sa charge de fonctionnement est plus importante donc sa probabilité de tomber en panne est alors aussi plus importante.

Nous donnons ici l'algorithme complètement détaillé, nous verrons que son implantation en annexe est très similaire.

3.2.3.a Messages et variables de l'algorithme

Pour l'essentiel, les messages ont la même structure et les mots-clés sont les mêmes que pour l'algorithme précédent excepté les mots-clés ***ouvre*** et ***maj*** qui sont supprimés. En contre-partie, un nouveau type de message est désigné sous le nom de ***jeton*** dont nous allons décrire l'utilisation.

Nous avons ajouté la variable id_frag qui se «substitue» à la variable racine. Elle contient l'identité de la composante (fragment) à laquelle appartient le nœud considéré. Les variables d'un site i sont alors les suivantes :

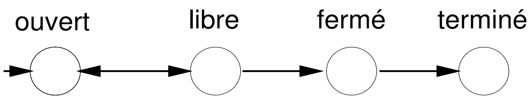

Figure 3.5 : États possibles d'un site

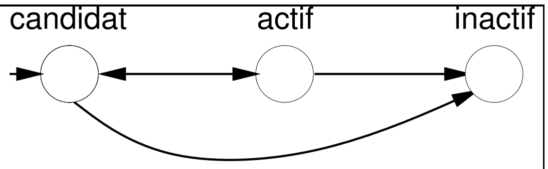

Figure 3.6 : États possibles d'un voisin

- mon_état : état courant du nœud, illustré par la figure 3.5
- id_frag : identité du fragment auquel appartient le nœud.
- père : identité de son père dans l'arborescence du fragment.
- fils : ensemble des identités de ses fils dans l'arborescence du fragment.
- état : tableau des états des voisins. L'automate de la figure 3.6 représente l'évolution de état$[x]$.

Les changements d'états de ces automates suivent les règles de l'algorithme précédent, paragraphe subsect:actcand.

3.2.3.b Description détaillée

Dans l'algorithme précédent, il y a en fait, en chaque fragment, deux nœuds particuliers : la racine (fixe) du fragment et le nœud muni du privilège d'émission du message ***conn***. Ici, ces deux nœuds sont confondus en un seul. Ainsi, le nœud muni du privilège *est* la racine et, l'identité du fragment n'est pas nécessairement celle de la racine. Par conséquent, notre algorithme présente des différences importantes avec le précédent, à savoir :

- Plus de mise à jour inconditionnelle,
- Des racines de fragment dynamiques.

Pour ce faire nous n'utilisons plus le message ***maj*** et nous ajoutons le message $< \_, \boldsymbol{jeton}, c, d >$: défini comme le message de passage du privilège « être racine du fragment ».

Au départ, tout site éveillé considère l'ensemble de ses voisins comme étant dans l'état `candidat`, il est donc lui-même dans l'état `ouvert`. Il n'a à cet instant ni père ni fils et l'identité de son fragment est égale à son identité, il est alors racine de son propre fragment.

Dans l'implantation que nous avons réalisée, nous avons affiné le choix du candidat dans la procédure **TENTE_FUSION**, ainsi que dans la procédure **PASSAGE_JETON**, par la gestion de candidats prioritaires – ceux à qui ont a répondu ***nok*** à une tentative de connexion. De plus, un site doit éviter de renvoyer le jeton vers un fils qui vient de lui envoyer, sauf si il ne peut pas faire autrement.

Algorithme **Distri-AC2**

```
INIT;
TANT QUE mon_état ≠ terminé FAIRE soit A soit B :
  [A] :                                                  (* émission *)
    SI racine ET attente_reponse = faux ALORS
      SI mon_état = ouvert
        ALORS SI ∄k, état[k]=candidat ET k ∉ fils
          ALORS PASSAGE_JETON ;
          SINON TENTE_FUSION ;
        SINON SI mon_état = inactif ALORS
          POUR TOUT i ∈ fils ENVOIE à i <fin> ;
          mon_état = terminé ;
  [B] :                                                  (* réception *)
    Soit EGO reçoit de y un message <jeton, state > ALORS
      fils ←fils ∪ {y} ; père ←nil ; MAJ(y, candidat) ;
    Soit EGO reçoit de y un message <conn, a > ALORS
      SI id_frag < a ALORS MAJ(y, candidat); ENVOIE à y <nok> ;
      SI id_frag = a ALORS MAJ(y, inactif); ENVOIE à y <cousin> ;
      SI id_frag > a ALORS FUSION(y) ; ENVOIE à y <ok, id_frag > ;
    Soit EGO reçoit de y un message <ok, id> ALORS
      attente_reponse ←faux ; id_frag ←id ; père ←y ;
      MAJ(y,inactif) ;
      POUR TOUT i ∈ fils ENVOIE à i <nrac, id> ;
    Soit EGO reçoit de y un message <cousin> ALORS
      attente_reponse ←faux ; MAJ(y, inactif) ;
    Soit EGO reçoit de y un message <nok> ALORS
      attente_reponse ←faux ; MAJ(y, actif) ;
    Soit EGO reçoit de y un message <nrac, id> ALORS
      id_frag ←id ; POUR TOUT i ∈ fils ENVOIE à i <nrac, id> ;
    Soit EGO reçoit de y un message <maj, state) > ALORS
      MAJ(y, state) ;
    Soit EGO reçoit de y un message <fin> ALORS
      POUR TOUT i ∈ fils ENVOIE à i <fin> ;
      mon_état ←terminé ;
FIN TANT QUE
```

Algorithme **Distri-AC2** (suite)

Procédure PASSAGE_JETON :

- Choisit $k \in$ fils tel que état[k] = `candidat` ;
- **MAJ**(k, `inactif`) ;
- père $\leftarrow k$; ENVOIE à k <***jeton***, mon_état> ;

Procédure TENTE_FUSION :

- Choisit k voisin de x tel que état[k] =`candidat` ;
- ENVOIE à k <***conn***, id_frag> ; attente_reponse $\leftarrow$VRAI ;

Procédure FUSION(y) :

- fils $\leftarrow$fils $\cup$ $\{y\}$; **MAJ**(y, `candidat`)

Procédure MAJ(y, state) :
    SI état[y] $\neq$ state ALORS
        SI $\nexists k$, état[k]=`candidat`
            ALORS SI $\nexists k$, état[k]=`actif`
                ALORS nouv_state = `fermé`
                SINON nouv_state = `libre`
            SINON nouv_state = `ouvert`
        SI nouv_state $\neq$ mon_état ALORS
            mon_état $\leftarrow$nouv_state ;
            SI père $\neq$ nil ALORS ENVOIE à père <***maj***, nouv_state>
FIN MAJ

3.2.3.c Émission et réception d'un message *conn*

Comme dans l'algorithme précédent, le message ***conn*** représente la tentative de connexion d'un fragment vers un autre. Trois cas se présentent : soit l'émetteur (y) a une identité de fragment strictement inférieure à celle du receveur EGO et dans ce cas il y a refus de fusion par le message ***nok***. Le deuxième cas est un autre cas de refus : l'égalité entre identité qui peut survenir comme dans la figure 3.7. Le dernier cas est le seul cas de fusion : lorsque l'identité de fragment de EGO est strictement supérieure à celle de y.

A la différence de l'algorithme précédent, le message ***conn*** ne peut être émis que par la racine du fragment sur une arête n'appartenant pas déjà au fragment.

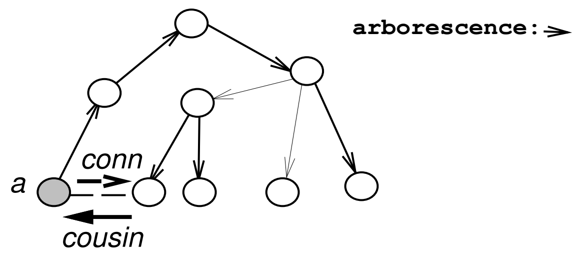

Figure 3.7 : Illustration du cousinage

3.2.3.d Émission et réception d'un message *ok*

Ce type de message a exactement la même fonction que dans l'algorithme précédent à savoir l'acceptation par un fragment d'identité supérieure à la fusion (absorption) d'un fragment d'identité inférieure. Ce message est une réponse au message ***conn***, il atteint nécessairement la racine du fragment émetteur puisque seul celle-ci a pu émettre le message ***conn***.

La réception d'un tel message provoque la mise à jour du tableau états et de mon_état comme précisé par les règles de l'algorithme précédent (voir le paragraphe subsect:actcand). Elle déclenche aussi une diffusion de messages ***nrac*** dans le fragment pour mettre à jour sur chaque site l'identité de celui-ci.

3.2.3.e Émission et réception d'un message *nok*

Au même titre que le message ***ok***, ce message est une réponse à un message ***conn*** et a donc les mêmes propriétés sauf qu'il représente une réponse négative donc une mise à jour différentes du tableau états et éventuellement de mon_état.

Dans le cas où mon_état devient `fermé`, alors la racine doit être déplacée. Le déplacement est réalisé par le message ***jeton***.

3.2.3.f Émission et réception d'un message *cousin*

Il s'agit du deuxième cas de refus de la tentaive de connexion. Ce message respecte donc les mêmes critères généraux que les deux précédents à savoir : ne circule que sur des arêtes hors fragment et le receveur est la racine du fragment qui a émis le message ***conn***. Ce dernier étant le message qui a provoqué l'émission de ***ok***, ***nok*** ou ***cousin***. Autre critère général : ce message provoque la mise à jour du tableau état et de mon_état.

Exactement comme pour le message ***nok***, cette mise à jour peut entrainer un déplacement de la racine. Il est à noter que ce message peut entrainer la terminaison de l'algorithme.

3.2.3.g Émission et réception d'un message *jeton*

Comme figuré par l'algorithme ci-dessus, le message ***jeton*** ne peut être émis que par la racine du fragment. Il est émis, lorsqu'il n'y a plus de voisin non-fils candidat, vers un fils candidat. Après avoir émis ce message EGO n'est plus racine du fragment : il a désormais un ancien fils : k comme père. k considère en recevant ce message qu'il est donc désormais racine du fragment et son ancien père (qui a émis le message) devient un fils. Il y a donc inversion du sens fils-père sur l'arête sur laquelle à circulé le message. Cette inversion donne lieu à une modification du tableau éat (et de mon_état) et est véhiculée dans les paramètres du message. Le receveur du message ***jeton*** considére donc son nouveau fils suivant ces paramètres.

3.2.3.h Analyse

Lemme 7 *L'algorithme ne génère pas de cycle et les fragments construits par l'algorithme sont connexes.*

Preuve : Les fragments construits par l'algorithme sont connexes car un fragment est construit uniquement par concaténation d'arêtes et de nœuds.

Les identité des processus induisent un ordre total sur les fragments car l'ensemble des identités des fragments est un sous-ensemble de l'ensemble des identités des processus. La règle d'acceptation (***ok***) d'un message ***conn*** induit une structure de demi-treillis fini. Le graphe d'un tel demi-treillis est sans cycle (voir [Berg70, Birk67] et l'exemple de la figure 3.2, page 68 : il n'y a pas de circuit créé car a refuse la demande de connexion provenant de c). □

Lemme 8 *Dans un fragment donné, il y a toujours un unique processus privilégié (qui est la racine du fragment) ou bien il n'y en a aucun et un message* ***jeton*** *est en cours de transmission sur une arête appartenant à ce fragment.*

Preuve : Par récurrence sur le nombre p de nœuds dans le fragment ; pour $p = 1$ la proposition est vérifiée car au départ tout processus est privilégié.

Si la proposition est vraie pour tout $i \leq p$ alors la proposition est vraie pour $p + k$ ($k \leq p$). En effet, supposons que le fragment F' de taille p envoie un ***conn*** via son processus privilégié i vers le processus j appartenant au fragment F de taille k et que $idcomp_i < idcomp_j$. j renvoie alors ***ok*** qui provoque sur F' une mise à jour d'identité et une perte de privilège. Comme il y a toujours un privilège dans le fragment F le fragment formé possède toujours un et un seul privilège ou bien ce même privilège est en transit pendant la construction via le message ***jeton***. □

Lemme 9*Pour toute arête (x, y) sortante d'un fragment, s'il n'y a plus de message en transit sur l'arête et si, en x, $Candidat[y]$ est égal à FAUX, alors, en y, $Candidat[x]$ contient VRAI.*

Preuve : Directement d'après la gestion des tableaux $Actif$ et $Candidat$: sur l'arête (x, y), lors de la réception de ***cousin*** ou ***ok***, l'arête n'est plus une arête sortante. Sur réception de ***conn***, soit il y a acceptation (***ok***) et l'arête devient une arête interne, soit il y a rejet (***nok***) et x n'est plus candidat pour y et inversement y devient candidat pour x. □

Lemme 10*Pour toute arête sortante (x, y), Si, en x, $Candidat[y]$ contient la valeur VRAI, alors, $Actif[y]$ contient la valeur VRAI.*

Preuve : Évident d'après les définitions. □

Lemme 11*Si pour le nœud racine d'un fragment, la valeur de la variable libre est FAUX et si il n'y a plus de message* ***conn*** *ou* ***jeton*** *en transit, alors l'algorithme est terminé et la réciproque est vraie.*

Preuve : Par définition, si *libre* contient FAUX cela signifie qu'il n'y a pas d'arête sortante, donc, que le fragment considéré est le seul restant (le graphe étant supposé connexe) et de ce fait, que l'on a construit un Arbre Couvrant. (S'il n'y a plus de message ***conn*** en transit alors la variable req contient FAUX ; Si il y a un message ***jeton*** en transit alors il n'y a pas de nœud privilégié dans le fragment).

La réciproque est évidente. □

Lemme 12*Si $Actif[y]$ est VRAI pour un processus x, alors au bout d'un temps fini $Actif[y]$ devient FAUX (et ne redevient jamais VRAI).*

Preuve : Supposons qu'il existe donc une arête (x, y) et que y soit actif pour x ; remarquons que d'après le lemme 1, il n'y a pas de circuit de créé, par conséquent aucun message ne peut circuler dans une boucle. Supposons que y soit candidat pour x (sinon d'après le lemme 9, en y, $Candidat[x]$ contient la valeur VRAI).

a) Si x est la racine de son fragment, il pourra envoyer un message ***conn*** vers y. Si la réponse est ***nok***, alors x devient candidat pour y tandis que y n'est plus candidat pour x, si la réponse est <***ok***> alors x n'est plus racine du fragment.

Enfin, si la réponse est ***cousin***, alors $Actif[y]$ est affecté à FAUX (et il n'y a aucun moyen de changer cette valeur).

b) Si x n'est pas la racine de son fragment, alors, son père est *ouvert* et le père de son père, ainsi de suite jusqu'à la racine du fragment. Puisqu'il n'y a pas de cycle et que d'après le lemme 5 l'algorithme ne peut pas être terminé, le nœud x sera donc examiné au bout d'un temps fini. C'est à dire qu'au bout d'un temps fini le nœud x deviendra à son tour racine de son fragment... □

Théorème 5*Si le réseau est connexe l'algorithme termine et construit un Arbre Couvrant.*

Preuve : Conséquence des lemmes précédents : lorsque le processus racine de l'Arbre Couvrant a ses variable *libre* et *req* à FAUX (il n'y a plus de message en transit), il envoie alors vers ses fils (qui le propagent) le message ***fin*** puis s'arrête. Si le réseau n'est pas connexe alors l'algorithme construit un Arbre Couvrant sur chaque composante connexe. □

3.2.3.i Complexité

Nombre de messages échangés Soit $N_{mot\_clé}$ le nombre total de messages portant le mot-clé ***mot_clé*** ayant circulé dans le réseau pour obtenir la terminaison de l'algorithme.

Les relations suivantes sont alors vérifiés :

$$\begin{aligned}
N_{fin} &= n-1; && (3.6)\\
N_{ok} &= n-1; && (3.7)\\
N_{cousin} &= m-(n-1); && (3.8)\\
N_{conn} &= N_{ok}+N_{nok}+N_{cousin}; && (3.9)\\
0 \leq N_{nok} &\leq \frac{n(n-1)}{2}; && (3.10)
\end{aligned}$$

Pour les équations (3.6), (3.7) et (3.8) ; $n-1$ est le nombre d'arêtes d'un Arbre Couvrant. Plus précisément, l'équation (3.8) provient du fait qu'une arête sur laquelle un message ***cousin*** a circulé n'est plus considérée dans le reste de l'algorithme et est exclue pour la construction de l'Arbre Couvrant (voir l'algorithme **BFS** en début de chapitre).

L'équation (3.9) est construite sur la remarque suivante : à tout message ***conn***, l'algorithme répond par un des trois messages suivants : ***cousin***, ***ok*** ou

nok. De plus ces derniers messages ne sont émis que sur réception d'un message ***conn***.

L'équation (3.10) contient un pire des cas ; le maximum étant atteint pour un graphe complet dont les messages sont ordonnés précisément de la façon suivante (l'algorithme étant message-driven on peut caractériser son exécution par une suite de messages) :

$$\left\{\begin{array}{ccc} n & \xrightarrow{\textbf{conn}} & n-1 \\ n-1 & \xrightarrow{\textbf{conn}} & n-2 \\ & \dots & \\ 2 & \xrightarrow{\textbf{conn}} & 1 \\ 1 & \xrightarrow{\textbf{conn}} & 2 \end{array}\right\} \quad \text{qui entraîne} \quad \left\{\begin{array}{ccc} n-1 & \xrightarrow{\textbf{nok}} & n \\ n-2 & \xrightarrow{\textbf{nok}} & n-1 \\ & \dots & \\ 1 & \xrightarrow{\textbf{nok}} & 2 \\ 2 & \xrightarrow{\textbf{ok}} & 1 \end{array}\right\}$$

puis,

$$\left\{\begin{array}{ccc} n & \xrightarrow{\textbf{conn}} & n-2 \\ & \dots & \\ 3 & \xrightarrow{\textbf{conn}} & 1 \\ 2 & \xrightarrow{\textbf{conn}} & 3 \end{array}\right\} \quad \text{qui entraîne} \quad \left\{\begin{array}{ccc} n-2 & \xrightarrow{\textbf{nok}} & n \\ & \dots & \\ 1 & \xrightarrow{\textbf{nok}} & 3 \\ 3 & \xrightarrow{\textbf{ok}} & 2 \end{array}\right\}$$

etc. jusqu'à

$$\left\{\begin{array}{ccc} n & \xrightarrow{\textbf{conn}} & 1 \\ n-1 & \xrightarrow{\textbf{conn}} & n \end{array}\right\} \quad \text{qui entraîne} \quad \left\{\begin{array}{ccc} 1 & \xrightarrow{\textbf{nok}} & n \\ n & \xrightarrow{\textbf{ok}} & n-1 \end{array}\right\}$$

d'où la borne maximum de l'équation (3.10). La borne minimale est atteinte par exemple pour un réseau en arbre dans lequel le père est toujours d'identité supérieure au maximum des fils et où tous les processus, sauf la racine, s'éveillent en émettant leur message ***conn*** vers leur père.

Les messages ***nrac***, sur le même exemple, sont au nombre de $(n-1)(n-2)/2$.

En conséquence, dans le pire des cas, l'algorithme est en $O(n^2)$. Ce relativement mauvais résultat est lié à l'absence d'une gestion par phases et donc à l'absence de périodes d'attente ainsi il présente un comportement difficile à étudier car à caractère « anarchique ». Nous verrons dans le chapitre 4 que son comportement en pratique est toutefois assez bon.

3.2.3.j Avec pannes

Nous allons expliquer maintenant comme rendre cet algorithme résistant aux pannes.

Tout d'abord remarquons que l'ajout d'une ligne ne pose pas de problèmes particuliers si les extrémités étaient au moins dans l'état actif ; dans le cas contraire, le lien est simplement marqué cousin.

Considérons le cas d'une panne franche de ligne. Si cette ligne correspond déjà à une ligne connectant deux cousins, aucune intervention n'est à entreprendre. Dans le cas contraire, elle est soit une arête sortante non encore testée soit elle représente un lien entre un père et son fils x.

Seul le second cas est à prendre en compte : deux fragments sont alors présents en lieu et place du premier. Il faut par conséquent un nouveau site privilégié pour le fragment de x, soit x lui même. L'identité de ce fragment doit être différente de celle du père ; de plus pour correspondre au fonctionnement de l'algorithme, l'identité de fragment ne peut qu'augmenter. Si on se contente d'incrémenter l'identité de fragment de x et de propager un ***nrac*** on risque de se retrouver avec un fragment ayant la même identité qu'un autre dans le réseau. Utiliser une notion de phase (en cas d'erreur le fils augmente sa phase de 1) diminue les risques mais ne les supprime pas. En effet, deux pannes à peu près simultanées sur le même fragment conduirait à 3 fragments dont deux aurait la même phase et la même identité.

Pour pouvoir distinguer entre le fragment fils et le fragment père, nous commençons par une « mise en panne » du fragment fils via une diffusion d'un message spécial. Cette mise en panne est en fait un simple marquage additionné au message ***nrac***. Une fois obtenu l'écho de cette diffusion, le site x va lancer une recherche d'une arête cousin le menant au fragment de son père ou à un autre fragment d'identité supérieure. Un site marqué envoie une tentative de connexion marquée, un site non marqué recevant une tentative de connexion marquée l'accepte dans les conditions habituelles mais aussi si les identités de fragment sont identiques. Un site marqué recevant une tentative de connexion marquée et de même identité la refuse par un message cousin usuel. Enfin, un site marqué recevant une tentative de connexion non marquée la rejette systématiquement, ceci pour éviter la formation de cycle. Le fragment étant ainsi isolé, le jeton du fragment de x finit soit par trouver un fragment dans lequel la fusion se fera, soit revient à x. Dans ce dernier cas, cela signifie qu'il n'existe plus de lien vers le fragment de l'ancien père. La mise en panne est alors supprimée et le comportement usuel reprend ses droits.

La période « d'isolement » (refus d'absorber d'autres fragments) du fragment est courte et n'introduit pas un retard trop conséquent, surtout de part l'hypothèse que les pannes sont des phénomènes assez rares.

La technique présentée permet d'assurer la constrution d'une forêt couvrante dans le cas ou le graphe des communications n'est plus connexe, plus précisément, l'algorithme construit alors un arbre couvrant dans chaque composante connexe.

3.2.4 Algorithme de croissance équilibrée

3.2.4.a Préliminaires

Nous reprenons les définitions proposées par E. Korach, S. Kutten et S .Moran dans [KoKM90] :

Un algorithme distribué A est *global* sur une classe Γ de graphes si pour tout graphe $G = (V, E)$ dans Γ, et pour toute exécution de A sur G, tout nœud ν de V soit reçoit, soit émet un message durant cette exécution. Il est évident qu'avec nos hypothèses de la section 1.4.5, tous les algorithmes d'élection et tous les algorithmes de construction d'arbre couvrant sont nécessairement globaux. Pour concevoir un algorithme d'élection (évidement pas d'arbre couvrant) non global il faut que chaque site dispose de connaissances précises sur le réseau comme, par exemple, le nombre d'initiateurs.

- Une exécution *monogénique*[3] (ou « rooted execution ») d'un algorithme A est une exécution dans laquelle un seul nœud s'éveille spontanément. Ce type d'exécution « centralisée » est utilisée par de nombreux algorithmes dans le domaine de l'algorithmique distribuée comme les algorithmes basés sur la notion d'écho, malheureusement tous les auteurs l'utilisant ne précisent pas toujours la nécessité d'une élection avant l'exécution de leur algorithme.
- Un algorithme A est *sériel* (ou « serial ») si pour toute exécution monogénique de A, à tout instant donné, au plus un message est envoyé dans le réseau. Cette notion est directement liée aux algorithmes ou réseaux dits à jeton, c'est à dire que tout site ne peut émettre de message que si il possède un (le) jeton, sinon il ne peut que recevoir des messages.
- Un *algorithme traversant* est un algorithme distribué qui est à la fois sériel et global. Le chemin suivi par les messages de cet algorithme forme alors, soit un chemin pseudo-hamiltonien, soit un cycle pseudo-hamiltonien. Par exemple, un algorithme de parcours de graphe en profondeur d'abord, en exécution monogénique, est un algorithme traversant.

L'algorithme proposé ici repose sur l'utilisation d'algorithmes traversants comme sous-procédures d'un algorithme maître. Les problèmes déjà évoqués sur les structures pseudo-hamiltoniennes sont donc valables ici aussi. Malgré tout, nous allons étudier cet algorithme car il présente un intérêt particulier : celui d'être modulaire et efficace en ce qui concerne le nombre de messages échangés

[3] Nous avons emprunté ce terme à la génétique et son sens premier est : issu d'un seul gène.

en pratique. Par modulaire, nous entendons ici le fait que cet algorithme pourra être rapidement adapté si la topologie du réseau vient à être connue par tous les sites. Cette information supplémentaire est en effet quelquefois disponible ; il est intéressant de pouvoir l'utiliser pour limiter la quantité d'information échangée. En théorie, seule l'adaptation de cette sous-procédure (que l'on appellera par la suite algorithme *porteur*) permet de particulariser l'algorithme complet suivant la topologie du réseau sur lequel il s'exécute. En pratique, pour des questions d'efficacité, il est préférable de fusionner les deux algorithmes comme dans l'algorithme proposé par E. Korach *et al.* dans le cas de l'élection sur un graphe quelconque.

Dans l'article [KoKM90], les auteurs utilisent les deux hypothèses suivantes :

- Liens bi-directionnels ou uni-directionnels, dont le type est connu par les processeurs les utilisant.
- La discipline FIFO sur l'ordre d'arrivée des messages n'est pas forcément respectée (utilisation d'un compteur).

Pour des raisons d'uniformité et de simplification de l'écriture de l'énoncé des algorithmes, nous avons choisi de conserver nos hypothèses de départ. En effet, si la discipline FIFO n'est pas respectée, les auteurs proposent d'utiliser un compteur d'arêtes traversées. Ce compteur est véhiculé par les messages et chaque site conserve le maximum sur les compteurs qui lui ont été donnés de voir. Cette construction permet, dans leur algorithme, d'ignorer les messages trop anciens.

Remarque : Une méthode générale, pour les cas où la discipline fifo n'est pas respectée est d'ajouter un compteur (local) de messages envoyés et de trier pour chaque ligne de communication les messages qui arrivent suivant leur numéro d'ordre.

La première hypothèse, quant à elle, nous semble discutable, dans un réseau réel, il est très peu pratique d'avoir des liens unidirectionnels (à part sur des topologies en anneau où des algorithmes spécifiques plus performants peuvent être mis en place). Un lien unidirectionnel « pur » c'est à dire où même un simple message d'acquittement ne peut circuler dans le sens contraire, est pratiquement inutilisable dès que le réseau n'est plus considéré comme parfaitement exempt de fautes.

3.2.4.b Messages échangés

Les messages échangés sont de type $<$*mot_clé*, *phase*, *domaine*$>$, où *mot_clé* est un élément de l'ensemble $\{$***annexion***, ***chasse***$\}$, *phase* est un entier, borné par n, indiquant la phase courante d'exécution. Le *domaine* est l'identité de l'initiateur de l'algorithme traversant.

L'algorithme présenté ici utilise un algorithme traversant (parcours de graphe en profondeur d'abord) comme support de diffusion de messages. Pour tout site i, les requêtes d'émission/réception de messages de l'algorithme maître utilisent les fonctions spéciales suivantes :

- InitB(w) : Initialise un algorithme porteur, d'initiateur i, portant le message w.
- ContB(w) : continue l'exécution normale de l'algorithme traversant avec w comme message porté.
- TermineB() : vraie si l'algorithme traversant est terminé.

Tout message reçu par un site est en fait reçu par l'algorithme porteur qui reconnaît alors si il s'agit d'un message de retour du parcours de graphe ou si il y a lieu d'informer l'algorithme maître. Comme pour les couches ISO évoquées dans le chapitre 2, l'algorithme porteur est indépendant de l'algorithme maître si et seulement si il n'a pas à lire le contenu du message porté (w). Là encore, en pratique, dans le cas où aucune information sur la topologie du graphe n'est disponible, les auteurs proposent, afin de diminuer le nombre de messages échangés, de procéder à une suppression logique d'arêtes « cousin » dans le parcours en profondeur d'abord, dans le cas des messages ***annexion*** de l'algorithme maître. Ces arêtes ne sont plus considérées par la suite par les autres exécutions de l'algorithme porteur. Cette astuce se révèle particulièrement efficace dans la pratique. Nous avons codé l'algorithme muni de cette amélioration et d'une terminaison par processus déterminant un arbre couvrant. Le code complet est placé en annexe.

3.2.4.c Variables

Les principales variables locales à un site sont *phase*, *dom*, *chass* et *candidat* qui contiennent respectivement la phase courante, le domaine courant, la phase éventuelle du jeton « chassé » et la phase d'un éventuel jeton en mode ***candidat***.

3.2.4.d Exemple sur un anneau

Nous allons étudier ici un cas particulier très avantageux pour cet algorithme : l'exécution sur un anneau. Supposons de plus que la topologie est connue, c'est à dire que l'algorithme porteur va être très simple et pour l'exemple, supposons même qu'il connaisse un sens de la direction sur cet anneau.

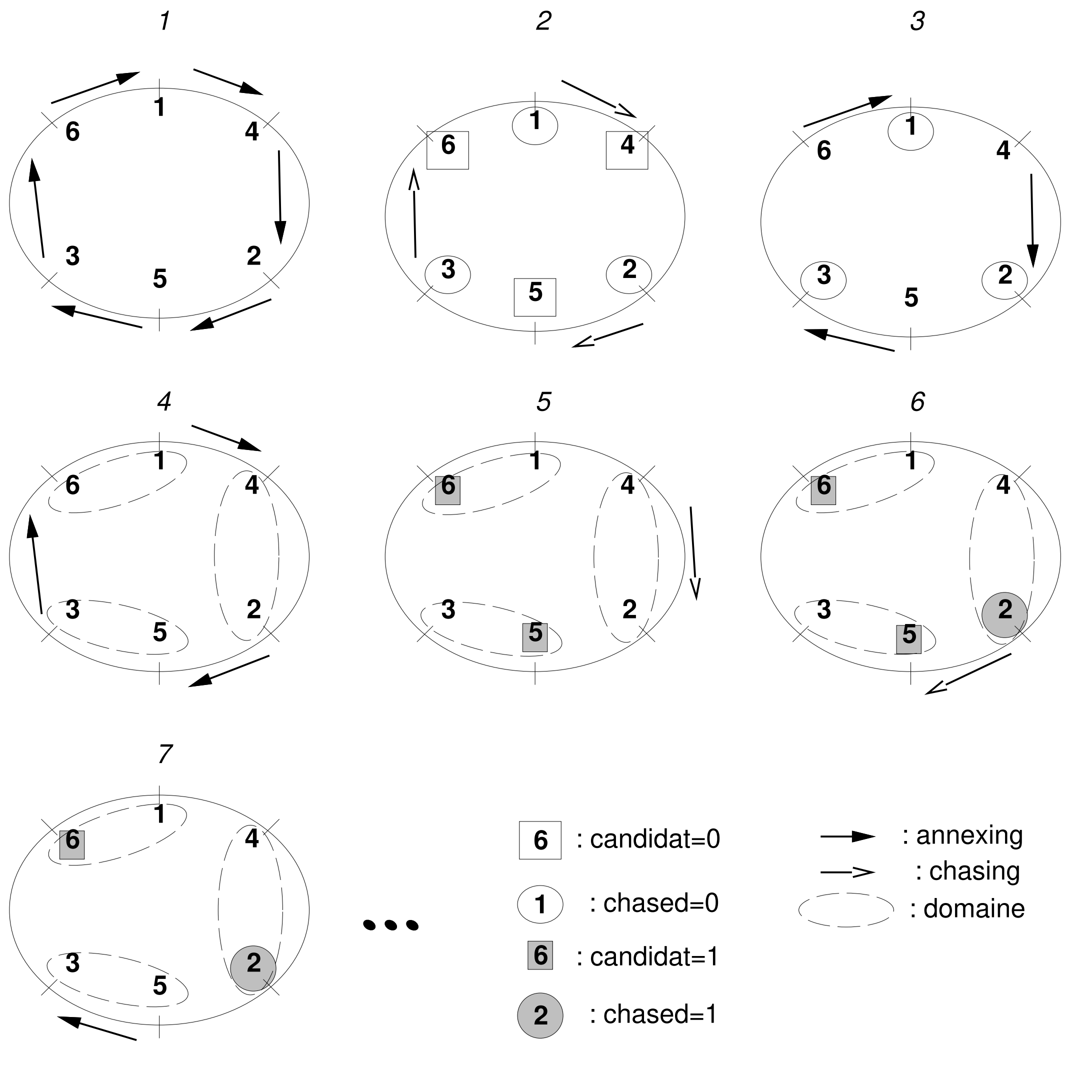

Figure 3.8 : Exemple d'exécution de [KoKM90]

Considérons la figure 3.8, pour que l'exemple ne soit pas trivial, nous avons choisi une numérotation alternée de l'anneau. Sur la figure, tous les sites démarrent l'algorithme à peu près en même temps, tous les fragments sont de phase 0 et réduits aux sites eux-mêmes. Nous allons supposer pour la simplicité de l'exposé que l'exécution est synchrone.

Dans l'étape *2*, le site 1 a émis un jeton $(0,1)$ en mode ***annexion*** et reçoit un jeton $(0,6)$. En conséquence, puisque $6 > 1$, il «part à la chasse» du jeton $(0,1)$ avec un message ***chasse*** et se marque lui-même chass(0). Dans le même temps le jeton $(0,1)$ est parvenu en 4 et devient un jeton candidat (et reste en 4).

Dans l'étape *3*, le site 4 reçoit le jeton ***chasse*** en provenance de 1 alors qu'il contient déjà un jeton en mode candidat d'où la destruction de ces deux jetons et la construction d'un nouveau de phase 1, émis vers 2. Dans cette étape, les sites 4, 5 et 6 passent en phase 1 et émettent de nouveaux jetons.

L'étape *4* illustre l'annexion, par un jeton en phase 1, des sites marqués par des jetons de phase inférieurs. Les messages sont alors propagés comme le stipule l'algorithme.

À partir de l'étape *5*, un seul message circule : il est en mode ***chasse*** pour les étapes *5* et *6*. Dans l'étape *7* le jeton en mode ***chasse*** rencontre un site contenant un jeton candidat. Il s'en suit que comme précédemment les deux jetons sont détruits au profit d'un nouveau de phase 2. Ce jeton fait le tour de l'anneau et reviends au site 5 qui pourra alors décider de la terminaison de l'algorithme et refaire un tour de l'anneau avec un message ***end*** pour assurer une terminaison par message.

3.2.4.e Algorithme sur des graphes quelconques

Nous allons reprendre ici les règles utilisées par l'algorithme d'élection sur des graphes quelconques proposé dans l'article de [KoKM90], simplifiées dans la mesure où nous considérons que la discipline FIFO est toujours respectée.

Chaque message est un jeton, disposant d'une phase et d'un domaine : (p, a) où p est la phase et a l'identité du site donnant l'identité du domaine.

Chaque jeton peut être dans trois modes distincts :

- mode ***annexion*** : un jeton dans ce mode tente d'annexer tous les sites du réseau. Pour ce faire, il utilise l'algorithme porteur pour parcourir le réseau, et il annexe au fur à mesure les sites rencontrés. Dans le cas où il revient au site a, et que l'algorithme porteur est terminé, l'algorithme est terminé car tous les sites appartiennent alors au même domaine et à la même phase.
- mode ***chasse*** : un jeton dans ce mode poursuit un jeton (p, b) $(b \neq a)$ en mode ***annexion***. Si il le rattrape alors un nouveau jeton, de phase $p+1$, est créé.
- mode ***candidat*** : un jeton dans ce mode est un jeton en attente sur un site. Il attend soit un jeton ***chasse***, soit un jeton ***annexion***. Là encore, si il est rejoint, un nouveau jeton de phase $p+1$, est créé.

Quand un jeton (p, a) rejoint un site c appartenant au domaine d'un jeton (q, b), l'un des deux algorithmes suivants s'applique en fonction du mode du jeton arrivant.

Algorithme **Mode Annexion**

Supposons ici que le jeton (p, a) est en mode ***annexion***.

A-1. Le jeton continue son parcours si et seulement si les deux conditions suivantes sont respectées :

(a) L'algorithme porteur n'est pas terminé.

(b) $q < p$ ou $(p, a) = (q, b)$ et le site c n'est pas marqué « chass(p) ».

Si $q < p$, alors le site c rejoint le domaine de (p, a).

A-2. Si l'algorithme porteur se termine (en annexant tous les nœuds) alors c est l'élu, sinon dans tous les autres cas le jeton (p, a) est détruit et une des conditions suivantes s'applique :

A-3. Si $p < q$ alors le jeton (p, a) est simplement détruit.

A-4. Si c contient un jeton en mode ***candidat*** en phase p, alors les deux jetons sont détruits et un nouveau jeton : $(p + 1, c)$ en mode ***annexion*** est créé qui suit alors le processus d'annexion.

A-5. Si $p = q$ et soit le site c est marqué « chass(p) », soit $b > a$ alors le jeton entre dans le mode ***candidat*** et reste en c.

A-6. Dans les autres cas (*i.e.* $p = q$, $b < a$ et le site c n'est pas marqué « chass(p) »), le jeton (p, a) est détruit et un autre jeton en mode ***chasse*** est créé qui va poursuivre (p, b).

Algorithme **Mode Chasse**

Supposons ici que le jeton (p, a) arrivant en c (appartenant au domaine de (q, b)) est en mode ***chasse***.

C-1. Le jeton continue son parcours si et seulement si les conditions suivantes sont respectées :

(a) $(p, a) = (q, b)$

(b) le site c n'est pas marqué « chass(p) »

(c) le site c ne contient pas de jeton en mode ***candidat*** et de phase p.

Si ces 3 conditions sont respectées alors le site c se marque « chass(p) » et relance le jeton. Dans les autres cas, le jeton est détruit et une des conditions suivantes s'applique :

C-2. Si $p < q$ alors le jeton (p, a) est simplement détruit.

C-3. Si c contient un jeton en mode ***candidat*** et de phase p, alors les deux jetons sont détruits et un nouveau jeton : $(p + 1, c)$ en mode ***annexion*** est créé comme en A-4.

C-4. Dans les autres cas (*i.e.* $(p, a) = (q, b)$ et un site c est marqué « chass(p) » ou $(p, a) \neq (q, b)$ et $p \geq q$), le jeton passe en mode ***candidat*** et attend en c.

3.2.4.f Analyse

Soit Γ une classe de graphes et $f(x)$ une fonction à valeurs réelles. On dira que Γ est *f-traversable* (*f arête traversable*) si il existe un algorithme traversant B tel que pour toute exécution monogénique de B, sur un graphe $G \in \Gamma$, et pour tout entier positif x, après avoir émis $\lfloor f(x) \rfloor$ messages, B doit avoir visité au moins $\min\{x+1, n\}$ nœuds distincts ($\min\{x+1, n\}$ arêtes distinctes). Ces $\min\{x+1, n\}$ nœuds distincts ($\min\{x+1, m\}$ arêtes distinctes) ont donc été utilisés dans l'émission / réception de ces $\lfloor f(x) \rfloor$ messages.

E. Korach *et al.* précisent dans [KoKM90] que toute classe de graphe est $O(x^3)$-traversable et $O(x^2)$ arête traversable. En fait, tout graphe est $O(x^2)$-traversable ($O(x)$ arête traversable) par exemple en prenant pour algorithme traversant B, un algorithme (distribué) de type parcours en profondeur d'abord. En effet, sur un graphe quelconque de taille t, une exécution mongénique de cet algorithme passe exactement deux fois par toute arête du graphe et par conséquent au plus $t^2 - t$ messages sont nécessaires à l'exécution complète de B sur ce graphe. Pour revenir à la définition précédente, après l'émission de x^2 messages, strictement plus de x nœuds ont été visités donc au moins $\min\{x+1, n\}$.

Soit $f(n)$ (ou $f(m)$) la fonction convexe représentant la complexité (dans le pire des cas) de l'exécution de l'algorithme traversant porteur.

Lemme 13 *Durant toute exécution de l'algorithme, le nombre de jetons distincts, créés à la phase p, est au plus de $k2^{-p}$, où k est le nombre d'initiateurs.*

Preuve : De par l'algorithme, tout jeton en mode ***annexion*** et de phase $p > 0$, est créé par la destruction de deux jetons de phase $p-1$. De plus, un jeton en mode ***chasse*** et de phase p est créé en détruisant un jeton en mode ***annexion*** et de même phase. □

Lemme 14 *Dans toute exécution de l'algorithme, au plus un nœud est élu.*

Preuve : Soit (p, a) le premier jeton qui déclare un nœud c (quelconque) comme élu. Ce jeton est par conséquent le premier qui ait réussi à visiter tous les nœuds via l'algorithme traversant porteur, ce qui implique qu'aucun de ces nœuds n'appartenaient au domaine d'un jeton de phase $q \geq p$, mais aussi qu'aucun jeton de phase $q \geq p$ ne sera créé. Quand aux jetons de phase strictement inférieure à p, l'algorithme traversant porteur de ce message ne pourra terminer. □

Lemme 15 *Durant une phase quelconque p, le nombre de messages circulant avec un jeton de phase p est au plus de $f(n) + n$.*

Preuve : Supposons que d jetons soient créés à la phase p, et que le domaine du i^{e}, $1 \leq i \leq d$, contienne n_i nœuds. Puisqu'aucun des deux domaines, à la même phase, ne se superposent, nous avons $\sum_{i=1}^{d} n_i \leq n$.

De par la propriété de l'algorithme traversant, le jeton (p, i) en mode ***annexion*** est porté par au plus $f(n_i)$ messages de cet algorithme. La convexité de f nous permet alors de majorer le nombre total M de messages portant des jetons en mode ***annexion*** :

$$M \leq \sum_{i=1}^{d} f(n_i) \leq f(\sum_{i=1}^{d} n_i) \leq f(n).$$

D'après l'algorithme, tout nœud qui émet un jeton en mode ***chasse*** et de phase p est aussitôt marqué « chass(p) » et il n'en émettra alors plus d'autre dans la même phase. Par conséquent, pour toute phase le nombre de messages portant un jeton en mode ***chasse*** est borné par n. □

Remarque : Précisons toutefois que la dernière assertion n'est respectée qu'à la condition d'implanter correctement l'algorithme combiné maître+porteur : il faut garder la trace du dernier message émis portant un jeton en mode ***annexion***

pour éviter des phénomènes de « retour-arrière » des jetons chasse, ce qui serait catastrophique pour le nombre total de messages échangés. Il peut malgré tout arriver que le jeton en mode ***chasse*** et celui qu'il « chasse » se croisent sur une arête d'où la nécessité d'un unique « retour-arrière » du jeton en mode ***chasse***.

Lemme 16 *Le nombre total de messages échangés durant une exécution de l'algorithme est* $(n + f(n))(\lg k + 1)$ *où* k *est le nombre d'initiateurs.*

Preuve : D'après le lemme 13, le nombre de phases est borné par $\lg k + 1$ et d'après le lemme 15 le nombre de messages par phase est borné par $f(n) + n$. □

Théorème 6 *Soit une classe de graphes* Γ f*-traversable où* f *est une fonction convexe*[4]*, alors il existe un algorithme d'élection dont la complexité en messages, sur tout graphe* $G \in \Gamma$*, est en au plus* $(n + f(n))(\lg n + 1)$.

Preuve : L'existence de l'algorithme de E. Korach *et al* en est la preuve ! □

3.2.4.g Complexité

Comme nous l'avons montré, les auteurs proposent une famille d'algorithmes dont la complexité en nombre de messages est en $[(f(n)+n)(\log k+1)$ ou $(f(m)+n)(\log k + 1)]$ où k est le nombre d'initiateurs et $f(n)$ $(f(m))$ est le nombre de messages nécessaires à l'algorithme traversant pour traverser les nœuds (arêtes) du réseau.

Si aucune information n'est disponible sur le type du réseau (graphe quelconque), ils obtiennent le meilleur résultat théorique actuel à notre connaissance qui est de $2m + 3n \log k + O(n)$ (si k nœuds démarrent l'algorithme) et à la condition d'une implantation non naïve de l'algorithme combiné maître+porteur où le porteur est un algorithme adaptatif de parcours en profondeur d'abord. Le code détaillé est exposé en annexe.

La complexité temporelle des algorithmes proposés est dans le pire des cas du même ordre que le nombre de messages échangés (comportement sériel), il est donc loin de l'optimalité en temps.

Nous étudierons, dans le chapitre 4, son comportement en pratique qui s'avère excellent.

[4] *i.e.* $f(x) + f(y) \leq f(x + y)$.

3.2.4.h Avec pannes

Nous allons proposer des modifications de l'algorithme en vue de le rendre résistant aux pannes franches de ligne.

Supposons la possibilité d'une coupure de ligne durant la construction de l'algorithme. Si cette ligne était déjà considérée comme reliant deux cousins dans l'algorithme porteur adaptatif, aucune opération spécifique n'est à entreprendre.

Dans le cas contraire, pour assurer la terminaison correcte de l'algorithme, nous proposons de lancer une nouvelle phase en chaque extrémité de l'arête en oubliant, en chaque extrémité, les éventuelles relation de cousinages déjà établies. Puisque nous nous sommes placés dans un contexte optimiste (voir la section 2.5), le nombre de pannes est faible et ainsi le nombre d'augmentation de phases restera faible. De plus, relancer un nouveau jeton à partir de chaque extrémité assure que si le réseau n'est plus connexe suite à la rupture de ligne, en chaque composante, un arbre couvrant est construit.

Il est évident que cette technique permet d'inclure les cas où un jeton se trouvait en cours de transmission sur l'arête défaillante. De même, nous incluons la possibilité de pannes simultanées en divers endroits du réseau.

Dans le cas ou l'algorithme a terminé ou est en phase de terminaison, la relance des jetons peut être conditionnée par le fait que l'arête appartient ou non à l'arbre couvrant final.

La panne franche d'un site a déjà été considérée via l'assimilation à la panne de toutes ses lignes de communication. Ceci permet d'adopter la technique d'augmentation de phase sur tous ses voisins.

Si l'on considère maintenant l'ajout d'une ligne de communication dans le cours de l'algorithme, nous pouvons, suivant certains cas de figure, soit adopter la technique précédente, soit ignorer cete ligne comme étant *a priori* un lien de type cousin. Cette deuxième méthode présente l'avantage de ne pas engendrer de messages mais a le défaut de ne pas assurer la complétion de la construction d'un arbre couvrant dans le cas ou le graphe pour être connexe a nécessairement besoin de l'arête ajoutée.

3.3 Algorithmes de construction d'AC de Poids Minimum

Dans les algorithmes qui vont suivre nous allons supposer que chaque site connaît, en plus des hypothèses générales faites au chapitre 1, les poids des arêtes menant vers ses voisins (et uniquement celles-là).

3.3.1 Algorithme « non déterministe »

Nous reprenons l'algorithme de I. Lavallée et G. Roucairol [LaRo86] dont nous avons traduit et très légèrement corrigé (quelques fautes de frappe et la ligne marquée *rectif.* a été simplifiée) le code, écrit à l'origine en CSP[5] non-bloquant, en pseudo-code comme le présente la page suivante.

Nous utilisons les notations (empruntées au langage LisP) $\mathsf{car}(\ell)$ pour le premier élément de la liste ℓ et $\mathsf{cdr}(\ell)$ pour le reste de la liste une fois le premier élément ôté.

3.3.1.a Démarrage

Au départ de l'algorithme, tout site éveillé est racine de son fragment et n'a donc ni fils ni père.

Il faut remarquer dans cet algorithme que les fusions sont telles qu'un fragment « absorbe » les fragments qui tentent de se connecter à lui à la condition que les demandeurs aient une identité (de fragment) plus grande.

[5] voir l'article de C. A. R. Hoare : [Hoar78].

Algorithme **Distri-ACPM1**

```
INIT ;
TANT QUE non terminé FAIRE soit A soit B :
[A] : SI racine <=EGO ET attente_reponse = faux ALORS
  CHOIX(a, b) ; attente_reponse = vrai ;
  x ← car(chemin(a)) ; chemin(a) = cdr(chemin(a)) ;
  ENVOIE à x <EGO, b, chemin(a),conn>
[B] :
  Soit EGO reçoit de y un message < id, dest, p,conn> ALORS
    SI dest =EGO ALORS
      SI id > racine ALORS
        ENVOIE à y < racine,ok> ; fils ← fils ∪ {y} ;
      SINON
        ENVOIE à y <nok> ;
    SINON
      x ← car(p) ; p = cdr(p) ;                    (* rectif. *)
      ENVOIE à x < id, dest, p,conn>
  Soit EGO reçoit de y un message < id,ok> ALORS
    SI racine =EGO ALORS
      racine ← id ;
      POUR TOUT x ∈ fils ENVOIE à x < racine,nrac> ;
      pred ← y ;
      ENVOIE à pred < maj, chemin > ;
    SINON
      ENVOIE à pred < id,ok> ;
      pred ← y ;
  Soit EGO reçoit de y un message <nok> ALORS
    SI racine =EGO ALORS attente_reponse ← faux ;
    SINON ENVOIE à pred <nok> ;
  Soit EGO reçoit de y un message < id,nrac> ALORS
    POUR TOUT x ∈ fils ENVOIE à x < id,nrac> ;
    racine ← id ; fils ← (fils ∪ {y}) − {pred} ;
  Soit EGO reçoit de y un message <maj, p > ALORS
    SI racine =EGO ALORS
      MAJ_ARBRE(p) ;
      SI succ = ∅ ALORS
        POUR TOUT x ∈ fils ENVOIE à x <fin>
        STOP.
    SINON
      ENVOIE à pred <maj, p >
```

La procédure **MAJ_ARBRE**(p) est locale, elle consiste en la fusion des arbres dont la racine est EGO avec l'arbre décrit dans p. Cette procédure reconstruit aussi l'ensemble des chemins de EGO vers les arêtes sortantes du fragment.

3.3.1.b Analyse

De par les mêmes remarques que celles utilisées dans le paragraphe 3.2.3.h pour la construction d'un Arbre Couvrant, l'algorithme ne construit pas de cycle.

De plus, son comportement étant similaire au premier algorithme étudié d'Arbre Couvrant étudié, il est clair qu'il termine en construisant un Arbre Couvrant. Cet Arbre est un ACPM à la condition de réaliser correctement la procédure **CHOIX** et la procédure **MAJ_ARBRE**(p), puisqu'en fait chaque racine gère la description de l'ensemble du fragment et de ses arêtes sortantes.

Cet algorithme comporte certains défauts : la longueur des messages est en $O(n)$ bits et le nombre total de messages est non borné dans le pire des cas. En effet, il est possible, si l'implantation de la fonction **CHOIX** n'est pas correctement réalisée, de procéder à tes tentatives répétées de connexion toujours vers le même voisin répondant invariablement ***nok***. On peut éviter ce comportement paradoxal en gérant une éventuelle attente dans le cas où tous les voisins extrémités d'arêtes sortantes de poids minimum répondent ***nok***. Il devient alors très similaire dans son comportement à l'algorithme de I. Lavallée et C. Lavault [LaLa89a].

3.3.2 Algorithme de croissance équilibrée

Il existe plusieurs algorithmes qui procèdent par croissance équilibrée des fragments, le plus connu et le plus efficace (puisqu'optimal en nombre de messages) est celui de R. G. Gallager, P. A. Humblet et P. M. Spira [GaHS83], que nous allons rappeler ici et dont nous donnons une version détaillée (et légèrement améliorée).

L'algorithme procède par fusion et absorption de fragments, suivant en cela les principes des propriétés que nous avons déjà définies. En particulier l'hypothèse spécifique principale de cet algorithme est la nécessité pour les poids des arêtes d'être distinctes, ce qui entraîne ici l'utilisation de la propriété 3 de la section 3.1.6.

Un fragment a une structure d'arborescence spéciale ; la « racine » n'est pas un nœud mais une arête : le cœur. L'orientation fils-père correspond à l'orientation vers ce cœur et le fragment est en fait deux arborescences reliées par le cœur. Les décisions de terminaison et de fusion sont prises par les deux nœuds extrémité du cœur.

3.3.2.a Mots-clés et structure des messages

Les messages sont de la forme <*identité*, *mot_clé*, *niveau*, *état*> où *identité* est l'identité de la composante à laquelle appartient le processeur émettant le message (poids de l'arête « cœur » du fragment) ; *mot_clé* est un élément de l'ensemble {***connexion***, ***init***, ***rapport***, ***test***, ***nrac***, ***accepte***, ***rejet***} et enfin, *niveau* représente le niveau du fragment émetteur.

Le message ***rapport*** est particulier : il doit véhiculer le poids de la meilleure arête sortante trouvée (en lieu et place du niveau).

3.3.2.b Règles de croissance des fragments

Chaque fragment recherche son arête sortante de poids minimum. Une fois l'arête $e = (i, j)$ trouvée, le fragment tente une fusion avec le fragment incluant le nœud j de l'autre extrémité de e. Nous distinguons la fusion de fragments et l'absorption d'un fragment par un autre, suivant les *niveaux* de chacun, comme suit :

Un fragment qui contient un et un seul nœud est de niveau 0. Supposons qu'un fragment F de niveau $L \geq 0$ est relié par son arête sortante de poids minimum $e = (i, j)$ avec un fragment F' de niveau L'. Si $L < L'$ alors le fragment F est *absorbé* par le fragment F' et le fragment ainsi créé a toujours la même identité et le même niveau. Si $L = L'$ et si F et F' ont la même arête sortante de poids minimum e alors il y a *fusion* des deux fragments pour en former un nouveau de niveau $L + 1$ et l'arête e est appelée *cœur* (core en anglais) du nouveau fragment. L'identité du fragment créé est donnée par le poids de son cœur : $w(e)$. Dans tous les autres cas, le fragment F doit attendre que F' grossisse suffisamment pour atteindre un niveau tel que l'une au moins des règles précédente s'applique.

Une **règle importante** est respectée tout au long de l'exécution de l'algorithme : l'identité d'un fragment ne change qu'avec son changement de niveau et vice et versa. De plus, de part l'hypothèse sur les poids distincts, les identités de fragment successives d'un même nœud forment une suite de valeurs distinctes.

La figure 3.9 illustre les règles précédentes : en grisé sont représentés deux fragments : F et F' de niveau 1. Le fragment F a comme arête cœur l'arête (1,2) ; le fragment F' a comme arête cœur l'arête (5,6). Supposons que pendant que le site 4 tente de se connecter au site 2, le fragment F et F' découvrent qu'ils ont la même arête sortante de poids minimum : la fusion des fragments est alors déclenchée, formant le nouveau fragment F" qui englobe F, F' et le site 4 qui était en attente d'une réponse. Il est possible sur ce même schéma que le site 4 soit absorbé par le fragment F avant la fusion de F et F', ce n'est qu'une question de temps.

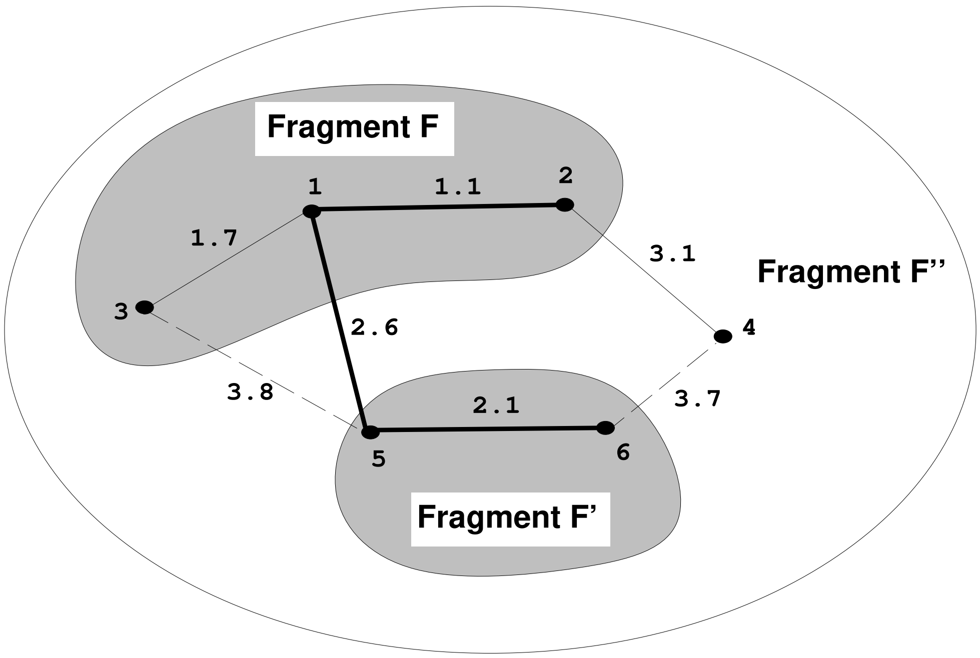

Figure 3.9 : Croissance équilibrée des fragments

3.3.2.c États possibles des nœuds et des arêtes

Un nœud peut être dans trois états distincts : `dort`, `trouve` ou `a_trouve`. Le premier état correspond à l'état initial, le deuxième est utilisé lors de la recherche d'une arête sortante de poids minimum et le dernier a d'autres instants (recherche terminée, fusion, etc.). Dès qu'il s'éveille (ou qu'il est réveillé par un événement extérieur ou par un message) l'état du nœud est positionné à `a_trouve` et ne retourne jamais à l'état `dort` durant l'exécution de l'algorithme.

Les arêtes adjacentes d'un nœud sont considérées comme étant soit `normal` (valeur initiale), soit `en_arbre` (appartenant à l'ACPM final), soit enfin `rejetée` (arête entre nœuds d'un même fragment). Une arête peut passer directement de l'état `normal` à l'état `rejetée`, mais une fois dans l'état `rejetée` ou dans l'état `en_arbre` elle reste dans cet état.

3.3.2.d Description de l'algorithme

Le code de l'algorithme qui suit utilise la commande **RENFILE_MSG** qui permet à un site de retarder l'interprétation d'un message en le replaçant en fin de tampon de réception.

Au départ, toutes les arêtes adjacentes à un site éveillé sont considérées dans l'état `normal`, sauf celle de plus petit poids qui est positionnée dans l'état `en_arbre`. Un site éveillé, soit spontanément (initiateur) soit sur réception de message, envoie immédiatement un message $<\boldsymbol{connexion}, 0>$ via l'arête dans l'état `en_arbre` avant même de lire l'éventuel message. La phase du site ainsi que son compteur *cpt_trouve* sont positionnés à 0 et son état à `a_trouve`. Dans le code qui suit, $w(\textsc{Ego}, y)$ désigne le poids de l'arête $(\textsc{Ego}, y)$ et représente l'identité du fragment.

Algorithme **Distri-ACPM2**

INIT ;
TANT QUE $mon_etat \neq$`terminé` FAIRE
  Soit $\text{E}\small{\text{GO}}$ reçoit de y le message $<\boldsymbol{connexion}, niv>$ ALORS
    SI $niv < niveau$ ALORS
      $etat[y] \leftarrow$ `en_arbre` ;
      ENVOIE à $y < idcomp, \boldsymbol{init}, niveau, mon\_etat >$;
      SI $mon\_etat =$`trouve` ALORS $cpt\_trouve \leftarrow cpt\_trouve + 1$;
    SINON SI $etat[y] =$`normal` ALORS
      RENFILE_MSG ;
    SINON
      ENVOIE à $y < w(\text{E}\small{\text{GO}}, y), \boldsymbol{init}, niv + 1, \texttt{trouve} >$
  Soit $\text{E}\small{\text{GO}}$ reçoit de y le message $< id, \boldsymbol{init}, niv, e >$ ALORS
    $niveau \leftarrow niv$; $idcomp \leftarrow id$; $mon\_etat \leftarrow e$;
    $pred \leftarrow y$; $meill\_arete \leftarrow 0$; $meill\_pds \leftarrow +\infty$;
    POUR TOUT i tel que $i \neq y$ ET $etat[i] =$ `en_arbre` FAIRE
      SI $e =$ `trouve` ALORS $cpt\_trouve \leftarrow cpt\_trouve + 1$;
    SI $e =$ `trouve` ALORS **TEST** ;
  Soit $\text{E}\small{\text{GO}}$ reçoit de y le message $< id, \boldsymbol{test}, niv, e >$ ALORS
    SI $niv > niveau$ ALORS
      RENFILE_MSG ;
    SINON SI $id \neq mon\_etat$ ALORS
      ENVOIE à $y <\boldsymbol{accepte}>$;
    SINON
      SI $etat[y] =$ `normal` ALORS $etat[y] =$`rejetée` ;
      SI $test\_arete \neq y$ ALORS
        ENVOIE à $y <\boldsymbol{rejet}>$;
      SINON
        TEST ;
  Soit $\text{E}\small{\text{GO}}$ reçoit de y le message $<\boldsymbol{accepte}>$ ALORS
    $test\_arete \leftarrow 0$;
    SI $w(\text{E}\small{\text{GO}}, y) < meill\_pds$ ALORS
      $meill\_arete \leftarrow y$; $meill\_pds \leftarrow w(\text{E}\small{\text{GO}}, y)$;
  Soit $\text{E}\small{\text{GO}}$ reçoit de y le message $<\boldsymbol{rejet}>$ ALORS
    SI $etat[y] =$ `normal` ALORS $etat[y] =$`rejetée` ;
    TEST ;

Algorithme **Distri-ACPM2** (suite)

```
                                                   (* Suite de la page précédente *)
  Soit EGO reçoit de y le message <rapport, poids > ALORS
    SI y ≠ pred ALORS
      cpt_trouve ← cpt_trouve − 1 ;
      SI poids < meill_pds ALORS
        meill_pds ← poids ; meill_arete ← y ;
      RAPPORT ;
    SINON
      SI mon_etat = trouve ALORS
        RENFILE_MSG ;
      SINON SI poids > meill_pds ALORS
        CHANGE_RAC ;
      SINON
        TERMINAISON ;
  Soit EGO reçoit de y le message <nrac> ALORS
    CHANGE_RAC ;
FIN TANT QUE
                                                                  (* Procédures *)
Procédure TEST :
  F ← {j ∈VOISINS / etat[j] = normal} ;
  SI F = ∅ ALORS
    test_arete ← 0 ;
    RAPPORT ;
  SINON
    Choisit(k ∈ F) ; test_arete ← k ;
    ENVOIE à k < idcomp,test, niveau, etat > ;
Procédure RAPPORT :
  SI cpt_trouve = 0 ET test_arete = 0 ALORS
    ENVOIE à pred <rapport, meill_pds > ;
Procédure CHANGE_RAC :
  SI etat[meill_arete] = en_arbre ALORS
    ENVOIE à meill_arete <nrac> ;
Procédure TERMINAISON :
  SI poids = meill_pds = +∞ ALORS
    POUR TOUS les i ≠ y ET etat[y] = en_arbre FAIRE(* Rectif. 1 *)
      ENVOIE à i <rapport, +∞ > ;                   (* Rectif. 2 *)
    mon_etat ←terminé
```

3.3.2.e Recherche de l'arête sortante de poids minimum

Considérons d'abord le cas trivial ou le fragment est de niveau 0, c'est à dire ou il ne comporte qu'un nœud unique. Dès que le nœud est réveillé (que ce soit par intervention extérieure ou par réception de message), il émet un message ***connexion*** sur son arête adjacente de poids minimum, marque cette arête `en_arbre` puisque ce sera une arête de l'ACPM final et passe dans l'état `a_trouve`, attendant la réponse à son message.

Pour les nœuds de niveau non-nul, nous allons considérer un fragment de niveau L qui vient de se former par la fusion de deux fragments (de niveau $L-1$). Les nœuds extrémités du cœur du nouveau fragment F commencent un nouveau cycle de recherche via la diffusion d'un message ***init*** suivant la relation père–fils (arêtes `en_arbre`). Ce message transporte la nouvelle identité du fragment, son niveau ainsi que l'état `trouve` déclarant ainsi la mise en place de la procédure de recherche en mettant tous les nœuds du fragment dans cet état. Si des fragments (au niveau $L-1$) sont attente de réponse à leur tentative de connexion, ils sont absorbés dans la foulée, les poussant eux aussi à la recherche de l'arête sortante de poids minimum. Il est d'ailleurs possible que d'autres fragments de niveau $L-1$ attendent des réponses de ces fragments (et ainsi de suite), ils sont alors eux-aussi absorbés dans le même élan.

3.3.2.f Réception d'un message *init*

Quand un nœud reçoit ce message, il définit son père comme étant l'émetteur du message et initialise ses variables locales suivant les valeurs des paramètres du message.

Si l'état porté par le message est `a_trouve`, le message est seulement relayé à ses fils car ce n'est alors qu'un message de mise à jour.

Dans le cas contraire (`trouve`), le message est aussi relayé à ses fils mais en plus localement il commence par chercher son arête `normale` de poids minimum (celles qui sont `en_arbre` appartiennent déjà au fragment, celles qui sont marquées `rejetée` sont des arêtes vers des nœuds du fragment). Sur cette arête, il envoie le message ***test*** avec son identité de fragment et son niveau. Dans le cas où aucune arête n'est disponible et que toutes les réponses sont revenues des fils, il envoie un message ***rapport*** vers son père avec le meilleur poids obtenu. Cette dernière remarque est valable pour les paragraphes suivants.

3.3.2.g Réception d'un message *test* via l'arête e

Sur réception de ce message, suivant les identités des fragments, deux cas se présentent :

- Si les identités des fragments sont identiques et que le message ne provient pas d'un père, un message ***rejet*** est émis et l'arête est marquée `rejetée`. Si le message provient d'un père alors le receveur r essaie à son tour d'envoyer un message ***test*** vers son arête sortante de poids minimum, exception faite toutefois dans le cas où r a déjà envoyé ce message via e : la réponse est alors un message ***rejet***.
- Dans le cas où les identités sont différentes, il faut considérer les niveaux L du message et LN du receveur : si $LN \geq L$ le message ***accepte*** est émis certifiant ainsi que l'arête est bien une arête sortante. Si $LN < L$ alors le message est retardé (replacé en queue de tampon) jusqu'à ce que le niveau du receveur soit suffisant. En effet, d'après la règle importante énoncée précédemment, si un fragment F' renvoie un message ***accepte*** comme réponse au message ***test***, l'identité de F' *doit* être différente *et le rester...*

3.3.2.h Réception d'un message *rejet* via l'arête e

L'arête e est marquée `rejetée` et la recherche d'arête sortante de poids minimum continue (voir la réception du message ***init***).

3.3.2.i Réception d'un message *accepte* via l'arête e

La recherche est fructueuse : si $w(e)$ est plus faible que le meilleur poids trouvé jusqu'à présent alors cette arête devient la meilleure. Un message ***rapport*** est renvoyé vers le père du receveur.

3.3.2.j Réception d'un message *rapport*

Deux configurations sont à considérer :

- Si le message ne vient pas d'un père (c'est qu'il vient d'un fils) alors on met à jour la meilleure arête trouvée jusqu'à présent jusqu'à temps que tous les fils aient répondu. Lorsque tous les fils ont répondu, un message ***rapport*** est émis vers son père.
- Si le message vient d'un père c'est qu'il vient en fait de traverser le cœur du fragment et donc qu'il contient la meilleure valeur trouvée jusqu'à présent dans l'autre partie du fragment. Cette valeur doit être comparée avec la meilleure

valeur obtenue dans l'autre arborescence. Il faudra donc retarder ce message tant que toutes les réponses des fils ne sont pas obtenues.

Le résultat de la comparaison permet d'obtenir la meilleure (au sens du poids) arête sortante du fragment ou bien l'absence d'arête sortante. L'absence d'arête sortante signifie que le fragment couvre la totalité du réseau et donc que l'algorithme est terminé.

Remarque : Afin d'obtenir une vraie terminaison (et non une terminaison par messages – voir le chapitre 1) nous avons dû ajouter les lignes commentées Rectif. 1 et Rectif. 2 dans le code. Ces lignes initialisent une diffusion du message ***rapport*** avec une valeur infinie comme meilleur poids troué. Cette diffusion est propagée par les nœuds extrémités de l'arête cœur du fragment vers tous les autres nœuds du graphe descendant l'arborescence du fragment final pour signaler que l'algorithme est terminé et donc que le retour à l'état `dort` (ou le passage à un autre algorithme) est possible.

3.3.2.k Analyse de l'algorithme

Deux fragments de niveau L se fusionnent si et seulement si ils ont la même arête sortante de poids minimum, formant ainsi un fragment de niveau $L + 1$. Sachant qu'au démarrage de l'algorithme tout nœud est un fragment de niveau 0, un fragment de niveau L contient au moins 2^L nœuds. Par conséquent, $\lg n$ est une borne supérieure sur le niveau maximum que peut atteindre un fragment.

De part les propriétés 2 et 3 du paragraphe 3.1.6, il suffit de vérifier que l'algorithme trouve bien les arêtes sortantes de poids minimum et qu'il n'y a pas d'inter-blocage pour montrer que l'algorithme est correct. La description détaillé de l'algorithme suffit à se convaincre que l'algorithme recherche bien les arêtes sortantes des fragments et qui sont de poids minimum.

Afin de prouver l'absence d'inter-blocages, considérons l'ensemble $\mathcal{F}$ des fragments existant à un instant donné, en omettant toutefois les fragments de niveau 0 qui ne consistent qu'en nœuds isolés passifs. En cet instant, si on suppose que l'algorithme a démarré mais n'est pas terminé, l'ensemble $\mathcal{F}$ est non vide et tout fragment de $\mathcal{F}$ à une arête sortante de poids minimum. Parmi les fragments de niveau minimum de $\mathcal{F}$ considérons celui dont l'arête sortante de poids minimum est celle ayant le plus petit poids. Tout message de ***test*** de ce fragment F réveille un nœud passif (de niveau 0) ou bien aura une réponse sans attente. De même tout message ***connexion*** soit réveille un nœud passif, soit rencontre un fragment de niveau strictement supérieur (qui entraîne la réponse immédiate d'un message ***init***) ou bien rencontre un fragment de niveau égal ayant la même arête sortante de poids minimum ce qui entraîne la création immédiate d'un fragment de niveau supérieur.

Nous avons testé l'algorithme sur plusieurs topologies et les résultats sont fournis dans le chapitre 4.

3.3.2.l Complexité

Dans [GaHS83], le nombre de messages échangés est annoncé en $2m+5n\log n+O(n)$; ceci à condition que les arêtes aient toutes des poids distincts.

En effet, une arête ne peut être rejetée seulement qu'une fois et chaque rejet requiert deux messages donc au plus $2m$ messages de type ***test*** et ***rejet*** sont échangés et conduisent à des rejets.

Ensuite, tant qu'un nœud appartient à un fragment de niveau différent du dernier et de 0, il peut recevoir au plus un message ***init*** et un message ***rejet***. Il peut transmettre au plus un message ***test***, un message ***rapport*** et un message ***nrac*** ou ***connexion***. Puisque le nombre de niveaux est borné par $\lg n$, dans le pire des cas, un nœud change successivement $\lg(n)-1$ fois de niveau (sans compter le niveau 0 et le dernier) ; ce qui conduit au total de $5n(\lg(n)-1)$ messages si l'on ne compte ni la première ni la dernière phase.

Au niveau 0, chaque nœud peut recevoir au plus un message ***init*** et peut transmettre au plus un message ***connexion***. A la dernière phase, chaque nœud peut envoyer au plus un message ***rapport***. Ceci conduit en tout à l'échange de $2m+5n\lg n$ messages.

Dans le même article, les auteurs exhibent un exemple où la complexité temporelle est en $O(n^2)$ si un seul processeur s'éveille spontanément. Dans le cas où tous les processeurs s'éveillent spontanément la complexité temporelle dans le pire des cas est en $O(n\log n)$. L'algorithme a été amélioré par B. Awerbuch dans [Awer87] qui obtient ainsi une complexité optimale à la fois en temps : $\theta(m)$ et en messages : $\theta(m+n\log n)$.

3.4 Algorithmes de construction d'AC de Diamètre Minimum

3.4.1 Un cas particulier

3.4.1.a Introduction

Un des problèmes majeurs est, dans le contexte de l'algorithmique distribuée, l'élaboration d'une structure de contrôle efficace couvrant la totalité du réseau.

Ce problème peut être résolu en construisant un Arbre Couvrant de Diamètre Minimal (ACDM) sur le graphe valué représentant le réseau des communications. Obtenir un arbre de diamètre minimal permet des gains substantiels sur les délais de transmission des messages de la plupart des algorithmes distribués car leur complexité temporelle est une fonction, au moins linéaire, du diamètre.

Après avoir précisé le problème qui, exprimé en ces termes, est nouveau, nous exposons la méthode de résolution en deux étapes. Nos premières études sur le sujet sont exposées dans [BuBu93, BuBu93b].

3.4.1.b Méthode

Nous présentons d'abord une méthode de construction d'un ACDM dans le cas d'un graphe valué uniquement avec des poids de 1, puis nous considérons le cas plus général de poids entiers positifs non nuls.

A partir des résultats déjà obtenus dans le chapitre 1 (paragraphe 1.3.2), $\rho(T) = \left\lfloor \frac{D(T)+1}{2} \right\rfloor$ pour tout arbre T.

Toujours dans le cas particulier où les poids sont tous égaux à 1, pour construire un ACDM, il suffit de choisir, parmi les Arbres des Plus Courts Chemins sur G (que l'on note $APCC(G)$), celui de diamètre minimum. Ce qu'exprime le lemme suivant :

Lemme 17 *Si* $w(u) = 1 \ \forall u \in U$, *alors* $min_{T \in APCC(G)} D(T) = D^*$.

Preuve : Considérons un ACDM T^*, et v_0 un centre de T^* ; Soit T un arbre des plus courts chemins de racine v_0 ; nous avons alors la propriété suivante :

$$\forall i \in X, d_T(i, v_0) = d_G(i, v_0) \leq d_{T^*}(i, v_0)$$

Donc le diamètre de T est inférieur ou égal au diamètre de T^* ce qui prouve le lemme précédent. □

Donc, pour construire un ACDM, il suffit de choisir parmi les APCC celui dont le diamètre est minimum.

Remarque : Dans ces conditions, un nœud donnant un APCC de diamètre minimum est un des centres du graphe. Mais il faut préciser que tous les centres du graphe ne donnent pas forcément des ACDM, car les arbres des plus courts chemins issus d'un nœud n'ont pas tous le même diamètre (à 1 près).

Cette remarque nous donne l'algorithme simplifié suivant qui construit un ACDM sur un graphe dont les poids des arêtes sont tous égaux à 1.

Algorithme **ACDM1**

1. Calculer l'ensemble $C(G)$ des centres du graphe,
2. Pour chaque centre v_0 calculer le diamètre d'un $APCC(v_0)$
3. Le centre donnant le meilleur diamètre donne l'ACDM.

Remarque : Un ACDM, dans le cas où les poids sont tous égaux à 1, a comme diamètre 2ρ ou $2\rho - 1$. Donc, dans la recherche du meilleur $APCC(v_0)$, on peut s'arrêter si la borne inférieure $(2\rho - 1)$ est atteinte.

Une implantation possible de cet algorithme passe par le calcul préalable des plus courts chemins. Tarjan a proposé [Tarj83] une implantation qui permet d'obtenir dans le cas présent une complexité temporelle en $O(m + n)$, ensuite il faut, pour chaque nœud, estimer le diamètre de l'APCC issu de ce nœud. Cette dernière étape est aisément réalisable en $O(n^2)$.

3.4.2 Algorithme de construction d'ACDM

3.4.2.a Introduction

Nous ne considérons ici que le cas où les poids des arêtes sont des entiers naturels (non nuls), ceci est assez proche de la réalité des réseaux de communication car, si l'on considère les poids des arêtes comme étant des délais moyens de communication point à point, la précision de la mesure n'est jamais infinie et donc on peut toujours se ramener au cas ou les poids sont des entiers naturels. En fait, les délais de communication sont assez variables dans la réalité et il faudra plutôt utiliser des délais moyens et, par conséquent, un très faible niveau de précision. Nous étudierons le problème de l'utilisation de poids réels ultérieurement.

La méthode de construction décrite est appliquée pour produire deux algorithmes : un algorithme séquentiel et un algorithme distribué présenté plus loin.

La difficulté du problème de construction d'un ACDM vient de la remarque suivante :

Remarque : Lorsque les poids des arêtes sont des entiers naturels quelconques, il existe des graphes G tels que le diamètre d'un ACDM est plus petit que le plus petit des diamètres des arbres des plus courts chemins $(\min_{T \in APCC(G)} D(T) > D^*)$.

Cette remarque reste vraie si l'on considère l'ensemble des arbres de poids minimum en lieu et place de l'ensemble des arbres des plus courts chemins. Il nous suffit de considérer l'exemple de la figure 3.10 : sur cet exemple, l'ACDM n'est ni un arbre des plus courts chemins ni un arbre de poids minimum.

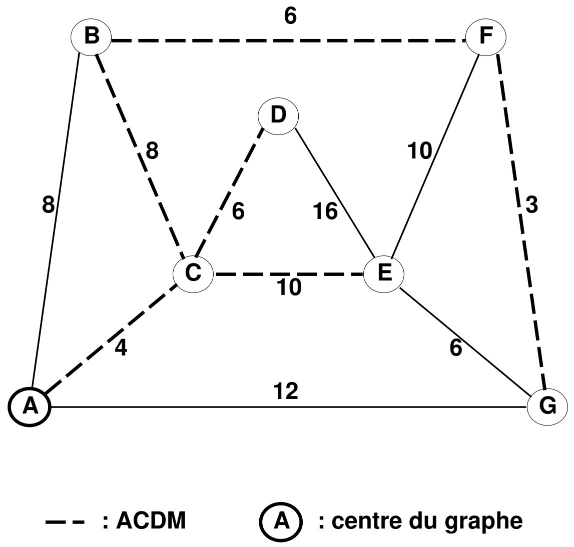

Figure 3.10 : Exemple d'Arbre Couvrant de Diamètre Minimal ; $D(G) = 22,\ D^* = 27$.

3.4.2.b Méthode

Pour construire un ACDM, une idée simple est de transformer le graphe G en un graphe G' valué uniquement avec des poids égaux à 1. Construisons donc le graphe $G' = (X', U')$, à partir de G, de la façon suivante : transformons les arêtes $e = (i, j)$ de poids p supérieur à 1 en p arêtes de poids 1 et créons $p - 1$ nœuds de sorte que le chemin de i à j dans G' ait une longueur p. Les arêtes de poids égal à 1 sont conservées.

Nous obtenons alors un graphe sur lequel on peut appliquer l'algorithme **ACDM1** précédant. Nous allons chercher à améliorer cette méthode en limitant le nombre de nœuds générés (que nous appelons nœuds fictifs par opposition aux nœuds réels que sont les nœuds de G).

Remarquons d'abord qu'il n'est pas nécessaire de calculer les diamètres de tous les APCC de tous les nœuds de G'. En effet, il nous suffit de calculer les diamètres des APCC des nœuds de G' vers les nœuds de G. De plus, le diamètre d'un APCC d'un nœud fictif d'une arête (i, j) est fonction uniquement des plus courts chemins dans G à partir de i et de j. C'est pourquoi notre algorithme commence par le calcul de toutes les distances $d(i, j)$.

Pour limiter l'exploration des nœuds fictifs, nous pouvons éliminer les arêtes les moins intéressantes. A cette fin, nous devons borner le diamètre D^* d'un ACDM sur G. La borne inférieure est évidente : le diamètre de l'ACDM est a *fortiori* au moins égal au diamètre du graphe.

Pour simplifier la suite de l'énoncé, $APCC(x)$ étant un ensemble, nous notons $D\_APCC(x)$ le plus petit diamètre des arbres de $APCC(x)$.

Nous allons considérer comme borne supérieure le plus petit des diamètres des arbres des plus courts chemins ($\min_{T \in APCC(G)} D(T)$).

Pour calculer cette borne, il nous faut calculer $D\_APCC(x)$ pour tous les nœuds x de G. L'algorithme suivant calcule cette valeur (ou une borne supérieure) pour x donné en utilisant les $d(x,k)$ et un tableau $tr(x,k)$ représentant la table de routage de x. La table de routage $tr(x,k)$ donne le voisin de x par lequel on obtient un des plus courts chemins pour aller de x vers k.

Algorithme **DIAM-APCC**(x)

Soient k_1 et k_2 les nœuds les plus éloignés de x
(tels que $d(x,k_1) \geq d(x,k_2)$).

SI $tr(x,k_1) \neq tr(x,k_2)$ ALORS
    DIAM-APCC$\leftarrow d(x,k_1) + d(x,k_2)$
SINON soit $y \in X$ tel que $tr(x,k_1) \neq tr(x,y)$
        et tel que $d(x,y)$ soit maximum.
    Soit $t \in X$ tel que $t = tr(x,k_1) = tr(x,k_2)$.
    SI $d(x,k_1) + d(x,k_2) - 2d(x,t) > d(x,k_1) + d(x,y)$ ALORS
        (* $D\_APCC(x) \leq d(x,k_1) + d(x,k_2) - 2d(x,t)$ et
        $D\_APCC(t) \leq D\_APCC(x)$ *)
        DIAM-APCC$\leftarrow d(x,k_1) + d(x,k_2) - 2d(x,t)$ (* 1 *)
    SINON (* $d(x,k_1) + d(x,k_2) - 2d(x,t) \leq d(x,k_1) + d(x,y)$ *)
        DIAM-APCC$\leftarrow d(x,k_1) + d(x,y)$.
    FSI
FSI

Dans le cas où l'algorithme s'arrête à la ligne marquée 1, cela signifie qu'il existe t tel que tous les arbres $T \in APCC(t)$ ont un diamètre inférieur a $D\_APCC(x)$. D'autre part, **DIAM-APCC**(v) est calculé pour tout $v \in X$ donc a *fortiori* **DIAM-APCC**(t) sera calculé et donnera une borne supérieure au moins aussi intéressante que celle qui serait donnée par x, c'est pourquoi dans ce cas il n'est pas utile de calculer la valeur exacte de $D\_APCC(x)$.

Remarque : L'application de cet algorithme sur le graphe de la figure 3.10 donne un diamètre de 28 pour un APCC à partir du nœud A et cette valeur donne la borne supérieure sur le diamètre de l'ACDM. D'autres exemples montrent qu'une première impression liée à la prépondérance des centres n'est pas fondée dans le cas général. Clairement les centres du graphe ne sont pas forcément les nœuds donnant les meilleurs diamètres.

3.4.2.c Un algorithme séquentiel

L'algorithme suivant construit un ACDM sur un graphe valué positivement.

Algorithme **ACDMp**

1.Calculer les distances $d(i,j)$ pour tout couple i,j de nœuds de G.

2.Calculer la borne supérieure : $bornesup \leftarrow \min_{x\in X}$**DIAM-APCC**$(x)$

3.$S \leftarrow U$.

4.TANT QUE $bornesup > D(G)$ ET qu'il existe au moins une arête $e \in S$ (telle que e est non-éliminée) :

(a)Calculer en chaque nœud fictif de e le diamètre d de l'APCC de ce nœud vers les nœuds de G.

(b)SI $d < bornesup$ ALORS $bornesup \leftarrow d$.

(c)$S \leftarrow S \cup \{e\}$

5.Le nœud x (réel ou fictif) tel que **DIAM-APCC**(x)= $bornesup$ donne un ACDM.

Les arêtes (i,j) que l'on peut « éliminer » de l'énumération sont celles vérifiant les conditions suivantes :

• Si $d(i,j) < w(i,j)$. En effet, cela signifie qu'il existe un chemin de longueur inférieure pour aller de i à j qui n'utilise pas cette arête.

Parmi celles qui n'ont pas été éliminées par cette méthode, calculons le rayon de l'arbre obtenu pour un nœud fictif v de (i,j) à la distance a de i :

$$\rho(T(a)) = \max_{k\in X}(\min\{a + d(i,k), d(j,k) + d(i,j) - a\})$$

On peut minorer cette expression, sachant que $d(i,j) > a \geq 1$:

$$\rho(T(a)) \geq 1 + \max_{k\in X}(\min\{d(i,k), d(j,k)\})$$

De plus, sur un arbre où les poids sont tous de 1, nous avons : $\left\lceil \frac{D}{2} \right\rceil = \rho$.

Donc, l'arête (i, j) peut-être éliminée dans le cas suivant :

•si $2 \max_{k \in X} (\min\{d(i,k), d(j,k)\}) + 1 >= bornesup$.

Sur notre exemple, cet algorithme n'explore que les arêtes (A,G) (pas d'amélioration), (B,C) qui donne l'ACDM de la figure 3.10 et (C,E) qui donne un autre ACDM (échange de l'arête (B,F) par l'arête (E,G)).

L'étape 1 de l'algorithme est en $O(mn \log_{(2+m/n)} n)$ si on choisit l'algorithme de calcul des plus courts chemins de Tarjan [Tarj83]. L'étape 2 est aisément réalisée en $O(n^2)$. Enfin, l'étape 4 consiste, dans le pire des cas, pour toute arête, à vérifier que l'on ne peut l'éliminer de l'énumération et ensuite de réaliser cette énumération en calculant les diamètres des $APCC(v)$ pour tout nœud fictif v.

Le calcul de $D_A PCC(v)$ où v est un nœud fictif d'une arête (x, y) et $x, y \in X$ peut se calculer avec l'algorithme **DIAM-APCC**(v) avec comme distances $d(v, z) = \min\{d(x, z) + d(x, v), d(v, y) + d(y, z)\}$.

En tout, cet algorithme séquentiel s'exécute en $O(mn(\log_{(2+m/n)} n + W))$ où W est le poids de l'arête de poids maximum. Clairement, cet algorithme peut être amélioré, mais ce n'est pas ici notre propos.

3.4.2.d Un algorithme distribué

L'algorithme suivant construit, en environnement distribué, un ACDM sur le graphe sous-jacent au réseau, les nœuds étant associés aux sites et les arêtes aux liens de communication comme pour les algorithmes de construction d'ACPM de la section précédente. Le poids d'une arête n'est initialement connue que par les deux sites extrémités de cette arête.

Algorithme **Distri-ACDMp**

- Calculer les $d(i,j)$ et $tr(i,j)$. *(* Soit UN le site d'identité minimale *)*

POUR TOUT SITE EGO :

1. $bornesup \leftarrow \min\{\textbf{DIAM-APCC}(\text{EGO}), 2\rho(G)\}$.
2. SI $upbound = D(G)$ ALORS ALLER EN 4.
3. POUR TOUTE arête $e = (\text{EGO}, j)$ telle que $j >$EGO et telle que l'arête e ne peut être ignorée FAIRE : *(* conditions définies en 3.4.2 *)*
   (a) Calculer le meilleur $d =$**DIAM-APCC**(x) où x est un nœud fictif de l'arête (EGO, j)
   (b) SI $d < bornesup$ ALORS $bornesup \leftarrow d$
4. Attendre des fils les messages avec leurs $bornesup$, envoyer le minimum vers UN.

- UN attend de recevoir les messages de tous ses fils avant de lancer la terminaison de l'algorithme.

Pour le calcul des distances $d(i,j)$ et des tables de routage $tr(i,j)$ associées, divers algorithmes de construction s'offrent à nous. Nous avons choisi d'utiliser l'algorithme de P. Merlin et A. Segall dans [MeSe79] parce qu'il est adaptatif et par là même résistant aux pannes.

Il a toutefois un défaut pour son utilisation dans notre algorithme : il n'assure qu'une terminaison par message. En effet, tant qu'il est utilisé comme algorithme de routage ceci ne pose pas de problème particulier, puisqu'un algorithme de routage doit, par définition, fonctionner de manière permanente.

L'algorithme de [MeSe79] nécessite la connaissance du nombre de sites du réseau. Cette condition est nécessaire dans la mesure où l'on veut un algorithme résistant aux pannes, si aucune panne n'est à craindre, il est plus intéressant, du point de vue de la complexité, d'utiliser un autre algorithme de routage. Nous développons ce sujet plus loin.

Cet algorithme est présenté pour un seul et même initiateur, chaque phase de mise à jour du poids d'une arête entraîne une mise à jour de l'arbre des plus courts chemins en deux phases. De façon informelle, les phases se décrivent comme suit :

1. Le « puits » (initiateur) diffuse des messages de contrôle permettant de mettre à jour les distances des sites par rapport à l'initiateur. Ces messages suivent un

parcours en largeur d'abord. Sur les arêtes ou les messages se croisent il y a éventuellement mise à jour de la distance par rapport à la racine, ce qui entraîne une mise à jour de la structure d'arbre déjà établie.

2. Des messages de contrôle remontent de fils en père vers l'initiateur.

On peut considérer que dans le cas d'une construction *ex nihilo*, le premier arbre à construire est un arbre couvrant quelconque, par exemple celui donné par l'algorithme **Distri-BFS** en début de chapitre. Ensuite on déclenche un certain nombre de mises à jour jusqu'a ce que plus aucune modification ne soit constatée dans l'arbre ce qui se réalise très simplement avec un drapeau (un booléen) porté par chaque message de retour. Chaque site opère alors un ou logique sur les drapeaux de ses fils et renvoie le résultat vers son père. Nous obtenons ainsi une terminaison par processus. Après cette terminaison, si une arête venait à être altérée (changement de poids, suppression, ajout) il suffit de redéclencher l'algorithme de routage pour obtenir à nouveau une arborescence des plus courts chemins.

Pour obtenir tous les plus courts chemins, c'est à dire de tout site à tout autre, nous nous contentons de superposer n exécutions de l'algorithme, chaque site étant un initiateur.

Nous calculons, en même temps que les tables de routage, $D(G)$ et $\rho(G)$ (modifications diffusées en même temps que les chemins améliorants). A ce niveau de l'exécution de l'algorithme, l'ensemble des identités des sites du réseau est connu de chaque site. Par conséquent, uniquement dans le but de simplifier l'écriture de la suite de l'algorithme, nous pouvons identifier le site d'identité minimum comme étant le site UN, et nous considérons désormais UN comme étant l'élu (voir le problème de l'élection en 3.1.2). L'arborescence fournie par les $d(\mathsf{UN}, k)$ est utilisée comme arborescence couvrante pour limiter le nombre de messages échangés, organiser la recherche de la borne supérieure et décider de la terminaison de l'algorithme.

Il est à noter qu'à la ligne 3a de l'algorithme, le site j doit coopérer pour que le calcul de d. Cette coopération s'exécute comme suit : à la demande de EGO, j envoie sa table de routage $(tr(j,k)\ \forall k)$ à EGO. Après réception de cette table, le site EGO peut calculer localement le meilleur d.

3.4.2.e Analyse de l'algorithme distribué

Nous allons analyser l'algorithme **Distri-ACDMp** du point de vue de la quantité d'information échangée plutôt que du point de vue du nombre de messages. La quantité d'information échangée est la somme des nombres de bits des messages échangés. Les messages doivent (pour la plupart) comporter l'identité de l'émetteur ainsi qu'un poids. Nous notons T la taille (en bits) d'un message ici égale à $O(\lg W + \lg n)$ où W représente le poids maximum d'une arête.

L'algorithme commence par la construction des plus courts chemins (calcul des $d(i,j)$) et propagation du rayon et du diamètre du graphe. Si l'on connaît une borne N sur le nombre de nœuds du réseau, cette étape, avec l'algorithme de P. Merlin et A. Segall [MeSe79], est exécutée avec $O(mn^2T)$ bits échangés. L'étape 1 ne comporte qu'un calcul local à chaque site. L'étape 4 est équivalente à la recherche de minimum sur un arbre de n nœuds et est donc en $O(n)$ messages. Dans l'étape 3a, l'algorithme effectue l'analyse de l'arête (EGO, j), $j >$EGO, en chaque site EGO. Cette exploration nécessite les tables de routage de EGO et de j. Suite à la demande de EGO, j envoie sa table de routage ainsi que les $d(j,k)$. Dans le pire des cas, la quasi totalité des arêtes est explorée. Finalement l'étape 3a provoque l'échange de $O(m)$ messages de $O(nT)$ bits. Enfin, la diffusion du message de terminaison ne représente que $O(nT)$ bits pour que tous connaissent D^* et le nœud (réel ou fictif) permettant sa génération.

En tout, l'algorithme **Distri-ACDMp** a la même complexité que l'algorithme de calcul des plus courts chemins : $O(mn^2T)$ bits échangés.

3.4.2.f Avec ou sans panne

En cas de panne franche de ligne (ou modification de son poids ou encore de suppression de cette ligne), la reconstruction de l'ACDM passe par la reconstruction des tables de routages, or, l'algorithme de [MeSe79] est adaptatif donc résistant à ce type de panne sans entraîner une reconstruction complète. Dans le meilleur cas l'arête impliquée n'appartient pas à l'ACDM et est ignorée dans les conditions définies au paragraphe 3.4.2, dans les autres cas, il faudra recommencer la construction à partir de l'étape 1, soit $O(mnT)$ bits échangés dans le pire des cas.

Si aucune panne ni évolution des poids des arêtes est à craindre, il est possible d'améliorer la complexité de l'algorithme présenté en choisissant un autre algorithme de calcul des plus courts chemins. Si l'on considère l'adaptation distribuée de l'algorithme de Dijkstra [Dijk59] proposée par G. N. Frederickson [Fred85], $O(n^2)$ messages sont échangés pour le calcul des plus courts chemins vers un seul nœud. C'est à dire qu'en tout $O(n^3T)$ bits (car $mn < n^3$).

Si les poids des arêtes représentent les délais de communication entre arêtes, et si aucune panne n'est à craindre, nous pouvons reprendre notre algorithme **BFS** donné en début de chapitre. Il suffit d'ajouter, à chaque message, l'identité de la source émettant le parcours en largeur d'abord et la distance à cette source. On obtient ainsi un algorithme de calcul de tous les plus courts chemins en $O(mn)$

messages, soit pour l'algorithme de construction d'arbre couvrant de diamètre minimum $O(mnT)$ bits échangés.

3.4.3 Développements

Parmi les développements possibles de ce problème, il faut envisager l'utilisation de poids réels. P. M. Camerini *et al.* dans [CaGM80] ont prouvé que le problème de l'ACDM (sous le nom de «max-path»), dans le cas où les poids sont des réels quelconques, est NP-complet. Par contre, si l'on ne considère que des réels positifs le problème est toujours polynomial.

Pour résoudre ce problème, nous pouvons d'abord constater que pour tout arbre T, il existe un nœud v (appelé centre absolu) fictif ou non, tel que : $2e_T(v) = D(T)$. En effet, on peut, à partir d'un chemin diamétral de T, de longueur $D(T)$, calculer la position de v simplement par $D(T)/2$. Par conséquent le problème de recherche d'un ACDM est ici identique au problème de la recherche d'un Arbre Couvrant de Rayon Minimal, problème identique au problème de la recherche d'un centre absolu, voir à ce sujet le livre de N. Christofides [Chri75].

3.4.4 Conclusion

Nous avons présenté dans cette section une solution au problème de l'élaboration d'une structure de contrôle efficace (*i.e.* un Arbre Couvrant de Diamètre Minimum) sur un réseau de sites communicants.

Les intérêts d'une telle construction sont multiples, elle offre, entre autres, une structure minimale pour un routage efficace prenant en compte d'éventuels changements de topologie du réseau (addition de liens ou sites) et peut être adoptée comme protocole de restructuration après panne.

L'algorithme que nous avons présenté fournit une solution originale à un problème re-formulé dans un cadre nouveau, il est basé sur des principes de calcul distribué utilisant des messages asynchrones, de plus cet algorithme est adaptatif et résistant aux pannes et a la même complexité que l'algorithme de calcul des plus courts chemins.

De nombreux problèmes attenants sont en cours d'étude notamment la résolution en distribué du problème de l'ACDM de poids minimum qui est un problème NP-complet, voir le livre de M. R. Garey et D. S. Johnson [GaJo79] ainsi que l'article de J.-M. Ho *et al* [HLCW91] en ce qui concerne les extensions possibles de type diamètre borné et poids total borné, problèmes eux-aussi NP-complets.

3.5 Conclusion du chapitre 3

Nous avons présenté dans ce chapitre, la résolution de problèmes de construction d'arbres couvrants avec et sans contraintes.

Ces problèmes sont essentiels en algorithmique distribuée de contrôle, ils permettent d'établir des structures couvrant la totalité du réseau et ainsi d'en faciliter la gestion. Nous avons choisi de considérer une structure particulière : l'Arbre Couvrant. Nous avons montré dans l'introduction de ce chapitre qu'il apporte plus d'avantages que l'anneau virtuel qui est aussi une structure de contrôle couvrant l'intégralité du réseau.

Considérant le problème de la construction d'Arbre Couvrant, nous avons constaté que d'autres problèmes telles que l'élection, la recherche d'extremum et la terminaison sont réductibles à ce problème.

Nous avons décrit ici plusieurs algorithmes construisant des Arbres Couvrants suivant l'absence ou la présence de contraintes spécifiques. Que ce soit en absence de contraintes ou avec la contrainte de poids total minimum, nous avons constaté l'intérêt théorique de l'utilisation de phases logiques pour diminuer le nombre de messages échangés. L'analyse du pire des cas des algorithmes distribués sans notion de phase met en œuvre l'utilisation de cas particuliers d'exécution entraînant un grand nombre de messages échangés. Ce que ne peut saisir l'analyse, au vu du comportement très «anarchique» de ces algorithmes, c'est le comportement *en moyenne*. Ceci nous conduit à considérer une étude pratique de ces algorithmes sur différents réseaux que nous avons résumée dans le chapitre 4.

La contrainte de poids minimum est la plus connue et la plus étudiée des contraintes sur la construction d'un Arbre Couvrant, mais elle n'apporte pas toujours un intérêt immédiat.

Nous avons présenté une contrainte nouvelle : celle de diamètre minimal qui nous a conduit a l'élaboration d'algorithmes nouveaux. Une structure de contrôle ayant un diamètre minimal est d'un intérêt évident : tout algorithme distribué étant fonction au moins linéairement, aussi bien en messages qu'en temps, du diamètre du réseau de communication sur lequel il s'exécute. Ainsi, nous considérons qu'un Arbre Couvrant de Diamètre Minimal peut être adopté comme un avantageux protocole de restructuration après panne.

Chapitre 4

De la simulation des algorithmes distribués

4.1 Introduction

La CM-1 «connection machine» avait plus de 60 000 processeurs mais ce n'était que des processeurs à un bits qui exécutaient tous le même code de façon synchrone. La réalisation d'une telle machine est en fait bien plus simple que la réalisation d'une machine de 100 processeurs 100 fois plus puissants. De plus il est particulièrement difficile d'assurer un fonctionnement synchrone de cette machine, la gestion des phénomènes de dérive d'horloge devenant rapidement l'essentiel du temps utilisé par le système.

Les machines actuelles se tournent ainsi résolument vers une architecture distribuée et un fonctionnement asynchrone. Elles ont tout de même toujours besoin d'un autre réseau ou mécanisme de routage pour la résistance aux pannes, la surveillance système etc. De plus, l'algorithmique distribuée n'étant pas toujours d'une utilisation et compréhension aisées, le système d'exploitation de ces machines s'efforce de masquer l'architecture à l'utilisateur, lui amenant à penser qu'il travaille sur une machine parallèle à mémoire partagée et non une machine à mémoire distribuée.

4.1.1 Simulation ou implantation ?

Le mécanisme de gestion de la mémoire dite partagée virtuelle est performant pour la plupart des applications des machines parallèles, à savoir le traitement de matrices sous toutes ses formes. En fait ce mécanisme est considéré comme absolument nécessaire par la plupart des constructeurs et programmer en distribué une machine distribuée est alors curieusement impossible ! Il existe pourtant, sur la KSR1 par exemple, des librairies développées, par différents laboratoires, un peu partout dans le monde et qui permettent de programmer par échange de messages. Malheureusement il n'existe pas de primitive de bas niveau pour cette gestion donc ces librairies sont en fait des sur-couches sur le mécanisme de mémoire partagée virtuelle ! Autrement dit, utiliser ces librairies équivaut à échanger des messages en utilisant la mémoire partagée qui utilise elle même un mécanisme d'échange de messages au niveau matériel. Il s'agit donc, déjà, de simulation.

L'implantation sur machine distribuée réelle ou sur réseau est donc conditionnée par de multiples paramètres totalement indépendants. Il faut, pour pouvoir réellement tester un algorithme distribué, disposer à sa convenance de réseaux ou de machines distribuées ayant des possibilités de reconfiguration de leur topologie. Rares sont les machines proposant de telles possibilités.

De plus, il faut que le système d'exploitation soit a peu près stable or les constructeurs, poussés par la concurrence, sont pressés de sortir des machines aux performances théoriques de plus en plus grandes et la conception des systèmes d'exploitation passe très (trop) souvent au second plan.

Ensuite, nous voulons pouvoir mettre au point des algorithmes distribués avec un maximum de confort et dans ce cas seul un système centralisé peut, pour l'instant, nous fournir éditeur, débogueur, visualisations etc. Nous pensons résolument que tels le papier et le crayon face aux traitements de textes, la simulation existera toujours comme outil de développement que ces soit pour les algorithmes distribués ou pour l'aérodynamisme des voitures.

Enfin, l'écriture d'un simulateur d'algorithme distribué permet d'acquérir une expérience certaine sur les problèmes de terminaison, démarrage et validation, notions essentielles en algorithmique distribuée.

Nous présentons, dans ce chapitre, le simulateur–évaluateur d'algorithmes distribués asynchrones que nous avons réalisé sur des machines parallèles, dans le but d'explorer les performances et défauts, en pratique, de ces algorithmes. Cette analyse permet d'avoir une représentation du comportement en moyenne des algorithmes, comportement qui n'a pas été étudié en théorie – rien d'ailleurs ne prouve qu'une telle analyse est possible dans le cas d'algorithmes sans gestions de phases (voir le chapitre 3).

Nous verrons que ce simulateur a été développé pour être portable sur différentes machines y compris des ordinateurs séquentiels classiques.

Après avoir exprimé les contraintes et caractéristiques qui nous intéressent, nous discutons des intérêts et inconvénients des simulateurs d'algorithmes distribués.

Nous avons implanté, grâce à ce simulateur et son langage très simple et très proche du pseudo code, les algorithmes de construction d'arbre couvrant présentés dans le chapitre 3 ainsi que des algorithmes de routage et de parcours de graphe. Les résultats de ces simulations sont exposés en section 4.4 et en annexe.

4.2 Spécifications générales

Notre objectif initial était, et est toujours, de simuler le fonctionnement d'un algorithme distribué asynchrone sur un réseau quelconque. Cet algorithme doit appartenir à la classe des algorithmes, définis dans le paragraphe 1.4.2.a, qui respectent une symétrie de texte. Il est bien évident que par le jeu de quelques tests cette condition n'est pas très restrictive.

La topologie du réseau étant donnée à l'exécution, l'algorithme peut considérer ou pas la connaissance du réseau, le nombre de nœuds, d'arêtes, la connaissance des identités des voisins, etc.

La simulation doit bien entendu fournir en fin d'exécution une trace de l'exécution de l'algorithme distribué avec le plus possible d'informations en cas d'échec ou de fonctionnement anormal puisqu'une des fonctions essentielles d'un simulateur est de pouvoir rapidement, et avec un certain confort, développer et mettre au point des algorithmes distribués.

Enfin, et c'est là la difficulté majeure de la réalisation d'un simulateur, la simulation doit être aussi proche que possible de la réalité, c'est à dire qu'elle ne doit pas cacher les défauts de l'algorithme qui seraient révélés par une exécution réelle. Cette dernière condition ne peut que rarement être respectée dans l'absolu, sinon le simulateur serait parfait et une implantation réelle n'aurait plus d'intérêt que potentiellement dans le cas où le temps serait un facteur primordial.

En fait, si nous avons choisi de faire de la simulation plus que de l'implantation sur une machine distribuée ou un réseau, c'est que, comme le savent tous ceux qui ont eu maille à partir avec l'implantation de programmes parallèles, le gain temporel n'est ni évident ni immédiat et un bon simulateur vaut mieux parfois qu'une machine parallèle (ou distribuée) parfois difficile d'accès et au système d'exploitation pas toujours complètement stable et débogué !

La figure 4.1 décrit le schéma de principe d'un simulateur, le langage C n'est figuré ici qu'à titre d'exemple. Les seules données fournies par l'utilisateur sont la description de l'algorithme dans un langage spécifique et les paramètres d'exécution.

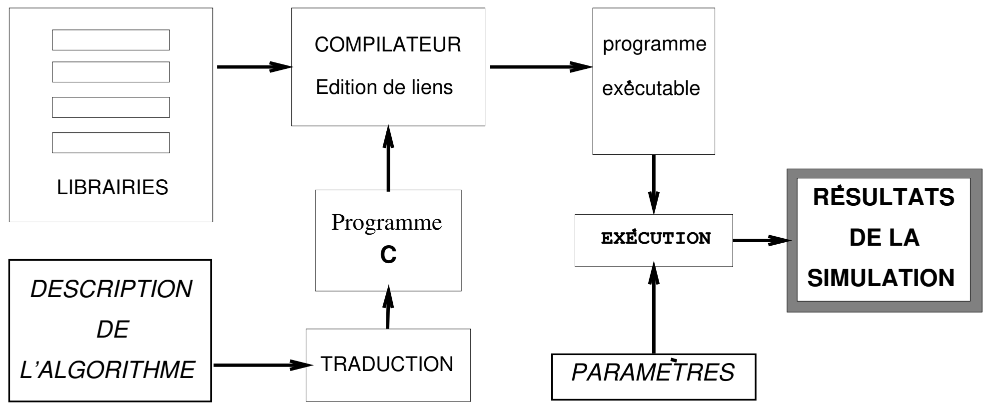

Figure 4.1 : Schéma de principe d'un simulateur

4.2.1 Intérêts de la simulation

La simulation est beaucoup moins coûteuse que l'implantation sur un réseau par exemple de transputeurs. En effet, le « deboguage » est facilité et réduit a une action centralisée à la fois sur les traces et sur le code. Bien plus encore que le déboguage, le simulateur assure un confort certain au développeur puisqu'il peut tester différentes topologies, parfois même sans avoir à recompiler le programme.

Une simulation d'un réseau de plus de 400 nœuds est possible mais l'accès à une machine distribuée d'une telle taille et en accès privilégié, c'est à dire ou il est possible que l'on ait effectivement autant de sites qu'en réclame l'algorithme, n'est pas toujours aisé.

4.2.2 Inconvénients de la simulation

De façon évidente, une simulation ne saurait être la réalité ; plus précisément, il est clair que nous n'obtenons par simulation qu'un sous-ensemble des comportements possibles dans le cas d'algorithmes distribués asynchrones.

Malgré tout, en intégrant des composants aléatoires nous pouvons produire des effets de retards sur des messages ou encore saturer artificiellement des tampons d'entrée/sortie. Finalement, il est possible dans certains cas d'obtenir la visualisation de défauts d'occurrence rare dans un système réel, voire des défauts qui ne surviendraient jamais parce que les machines distribués et les réseaux actuels disposent souvent d'une couche réseau câblée qui, d'elle-même, évite la transformation de certaines fautes en défaillances.

4.3 Le simulateur–évaluateur

4.3.1 Fonctionnement

Notre simulateur respecte le modèle de système distribué Séfini dans la section 1.4 et les hypothèses du paragraphe 1.4.5. Nous allons le décrire, caractéristique après caractéristique, avec une partie des techniques qui le constitue.

A partir du code d'un algorithme distribué découpé en étapes ou instructions, le simulateur exécute l'algorithme sur chaque site, de façon répétitive, instruction après instruction. Pour chaque étape, le programmeur doit préciser quelle est l'étape suivante à exécuter où bien l'arrêt définitif du site considéré.

En cela, il respecte la notion de séquentialité d'un site, un algorithme distribué étant l'expression des échanges de messages (et comportements internes) de sites séquentiels. Nous avons déjà discuté de la possibilité pour ces sites de représenter en fait des réseaux locaux. Cela est possible à la condition de définir dans ce réseau local un maître ou responsable des communications vers l'extérieur. Celui là est effectivement séquentiel que ce soit une machine ou un processeur. Cette hypothèse ne nous semble pas restrictive car elle provient de la simple observation des réseaux actuels.

Le programme se termine lorsque tous les sites ont déclaré avoir terminé ou, si l'utilisateur le désire, un site est déclaré inactif après un certain délai de non réception de message et par la même, une inactivité totale car nous ne cherchons à simuler que des algorithmes de type «message driven». Il existe en effet des algorithmes, la plupart des algorithmes de routage par exemple, qui ne sont pas réalisés en vu d'une terminaison par processus, il nous faut donc définir une limite à leur simulation. Le critère d'arrêt étant alors la terminaison par message, il nous suffira de compter, pour chaque site simulé, le nombre de fois qu'il cherche à défiler un message de son tampon de communication à partir de la dernière réception de message.

Dans le cas de la terminaison par processus, un site sait quand il a terminé, il suffit alors qu'il incrémente une variable partagée indiquant le nombre de sites ayant terminé. L'arrêt complet de la simulation est alors obtenu lorsque cette variable est égale au nombre de sites simulés. En limitant le délai autorisé d'inactivité nous pouvons estimer un «degré de parallélisme» c'est à dire la possibilité de mesurer si la répartition du travail est ou n'est pas équitable au vu du délai.

Clairement ceci sera dépendant de la machine sur laquelle se fera la simulation.

Pour chaque étape, le programmeur peut préciser si il veut une trace ou pas. Une trace étant un « snapshot » (cliché) de l'état instantané de tous les sites. L'état d'un site est défini par l'ensemble des valeurs des variables locales à ce site. Les traces ne sont activées qu'à la condition que l'utilisateur spécifie le paramètre correspondant lors de l'exécution.

Nous devons remarquer que l'observation d'un phénomène peu influencer ce phénomène. Ainsi en va de l'observation au microscope des bactéries vivantes de part la chaleur dégagée par l'éclairage. De même, l'observation de l'état d'un site simulé consiste en l'impression d'une trace sur fichier ou périphérique. Cet impression induit un retard dans l'exécution de ce site et ainsi perturbe l'ordre d'émission/réception des messages. De plus, l'impression sur fichier ou périphérique doit être associée à un verrou d'exclusion mutuelle ce qui introduit une séquentialisation de l'exécution. Pour plus de considérations sur ce sujet voir entres autres J. E. Lump *et al.* [LCGW+93] qui ont étudié l'influence de l'intrusion sur la mesure.

Notre simulation est une simulation modulaire sur un système centralisé parallèle. Par conséquent, il est facile de modifier des paramètres et de voir immédiatement (parfois sans recompilation) les comportements induits. Nous obtenons des traces soit globales, soit locales, de l'état des processus ainsi que la possibilité d'implanter des méthodes dynamiques de surveillance de l'algorithme : critères d'efficacité, nombre de messages d'un type donné, etc.

Le nombre de nœuds du réseau est un paramètre au même titre que la topologie et il peut être arbitrairement grand dans la limite de la mémoire disponible.

Nous avons choisi de construire un simulateur sur une machine parallèle et donc de le paralléliser pour rendre les exécutions plus rapides. Bien entendu, le nombre de processeurs physiques disponibles est bien inférieur au nombre moyen de nœuds des réseaux sur lesquels nous voulons travailler... On pourrait imaginer qu'il suffit de créer autant de processus sur la machine que de processus à simuler. Malheureusement, les machines auxquelles nous avons pu avoir l'accès ne disposaient que d'UNIX (ou de variantes) et les processus UNIX sont très gourmands en temps et en mémoire. Il est par exemple inconcevable pour cette machine de créer plus de 200 processus. Le concept de processus léger ou « thread » dans la norme Posix permettrait peut être de palier à ce défaut, malheureusement, nos essais sur la machine KSR1 ne sont pour l'instant pas très concluants. A l'heure actuelle il est tout à fait possible de créer 400 processus légers sur une KSR1, mais le fait d'exécuter plusieurs processus légers par un même processeur physique peut conduire à des pannes fatales de la machine (fonction *getsubpage()* de la librairie C).

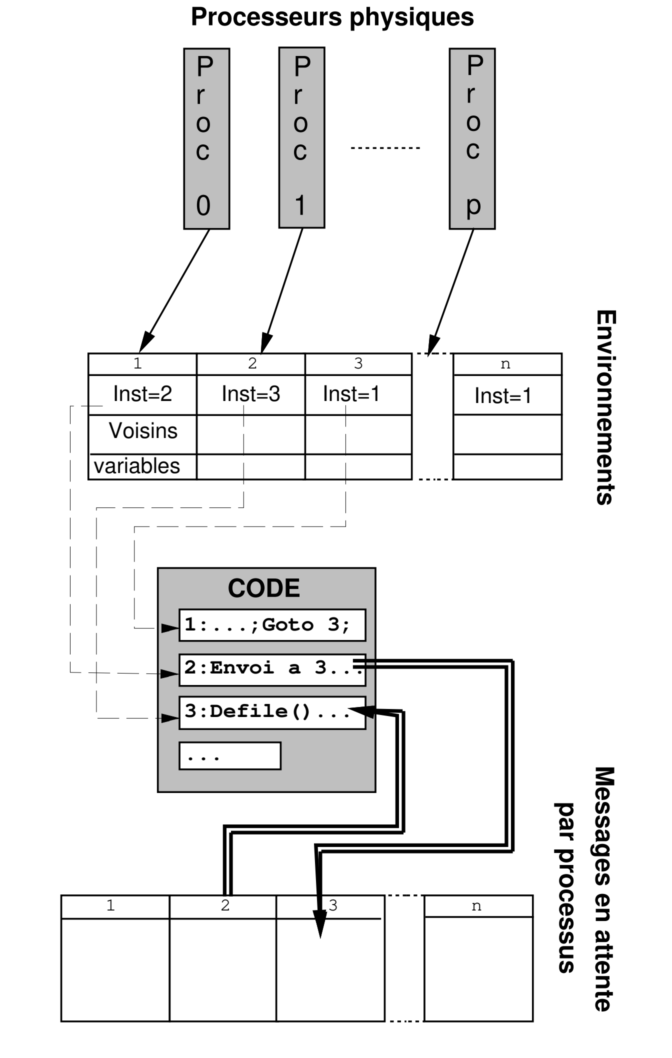

Figure 4.2 : Exécution en temps partagé parallèle

Par conséquent, nous créons seulement autant de processus que de processeurs physiques accessibles (a moins que dans une optique uniquement temps-partagé seulement quelques processus soient disponibles) et nous chargeons chacun d'exécuter seulement une étape d'un processus simulé, puis lorsque cette étape est exécutée, de chercher un autre site à simuler qui soit «libre» c'est à dire qui ne soit pas déjà en cours de simulation. Nous sommes donc dans un cas d'implantation d'exécution en temps partagé parallèle comme le montre la figure 4.2. Évidemment, si le nombre de processeurs physiques disponibles p est égal à 1, *i.e.* une machine SISD traditionnelle, il s'agit d'une exécution en temps partagé standard. Une légère amélioration du simulateur pourrait être de déléguer, dans ce cas, la charge de l'exécution en temps partagée au système d'exploitation lui-même, mais ce serait au prix d'une certaine perte de portabilité.

4.3.2 Langage de description

4.3.2.a Structure

Nous avons choisi de ne pas redéfinir un langage complexe de description, parmi les nombreux déjà existants, mais de se baser sur la connaissance minimale des utilisateurs, à savoir un langage de programmation très classique (nous avons choisi le langage *C* pour sa portabilité et le très grand nombre de compilateurs existants) auquel nous avons ajouté une librairie de fonctions les plus élémentaires possible, plus des facilitées de manipulation d'ensembles. En fait, l'idée principale est de rester proche de la description en pseudo-code tout en permettant une première vérification de syntaxe avant de transformer la description en un programme *C* indépendant de la machine. Ce programme peut alors être soit directement compilé sur la machine cible avec les librairies que nous avons définies plus la librairie spécifique à la machine. Dans des utilisations très particulières, ou pourra éditer au besoin le programme *C* précédent avant sa compilation.

L'algorithme suivant représente la structure de l'algorithme distribué de parcours de graphe en profondeur d'abord (dfs) avec un seul initiateur, codé pour notre simulateur. Ce type d'algorithme est utilisé par de nombreux algorithmes de construction d'arbre couvrant, notamment l'algorithme de E. Korach *et al.* [KoKM90]. Le site d'identité 1 est supposé être l'initiateur.

Algorithme **Distri-DFS** (pour le simulateur)

```
VAR
    Ensemble candidats, fils;
    Pidtype pere;
FIN
MCL
    dfs, dfs_back, cousin;
FIN

lance_dfs()
{
    Pidtype i;
        i=Ens_alea(candidats); Ens_ote(i,candidats);
        Envoi(i,dfs,""); Ens_ajoute(i,fils);
}
FIN

CALCUL                                              (* Le programme *)
    1 : Ens_copy(VOISINS,candidats);
        pere=0; Ens_nul(fils);
        if (MYPID== 1) lance_dfs();'(* Le site 1 est l'initiateur *)
        Va_en 2;
    2 : if (Defile()) BRANCH_ON MCL;
        break;
    ON_MESG dfs:
        if (pere==0 && MYPID!=1)
        {                              (* Le site Mypid pas encore visité *)
            pere=ORG; Ens_ote(ORG,candidats);
            lance_dfs();
        }
        else
        {                              (* Le site Mypid a déjà été visité *)
            Envoi(ORG,cousin,"");
            Ens_ote(ORG,candidats);
        }
        Va_en 2;
    ON_MESG cousin:
        ...
    ON_MESG dfs_back:
        ...
FIN
```

Le lecteur pourra remarquer un découpage en quatre blocs principaux : le premier, le bloc **VAR**, déclare les variables locales à chaque site ; le bloc **MCL** rassemble les mots-clés des messages (leur type) à des fins de comptage. Le bloc suivant est optionnel, il est réservé aux fonctions définies par l'utilisateur, chacune devant être (pour l'instant) terminée par **FIN** ce qui permet une analyse syntaxique simple. Un autre bloc optionnel, non représenté ici, est le bloc d'affichage utilisé lors des traces d'exécutions, il permet de préciser le format d'affichage de ces traces.

Le dernier bloc est le plus important, le bloc **calcul**, c'est le bloc de déclaration de la séquence des instructions qui forme l'algorithme en lui même. Le découpage en instructions est arbitraire, il est laissé à la discrétion de l'utilisateur, les seules contraintes imposées sont l'existence d'une instruction numérotée 1, qui est la première exécutée et l'absence d'instruction 0 qui est réservée à la déclaration de terminaison (commande **THE END**).

Toute fonction définie par l'utilisateur, ainsi que le bloc principal, est une fonction au sens du langage C, et peut utiliser les variables déclarées comme étant locales au site.

Des primitives de gestion émission/réception de message sont fournies : Envoi(x,mcl,msg) et Defile(). Le premier représente l'envoi d'un message de type mcl, portant le message msg, au site x. Le second teste si le tampon de réception de message est vide, dans le cas contraire, les variables spéciales ORG, MCL et MESG sont affectées respectivement à l'origine du message, son type (mot-clé) et le contenu du message.

4.3.2.b Variables et types de données

Le lecteur attentif remarquera l'utilisation d'autres variables non déclarées telles que MYPID, et éventuellement NBPID dans d'autres algorithmes des annexes, qui donnent respectivement l'identité du site et le nombre de nœuds du réseau initial. La variable VOISINS est un *ensemble* dans un sens que nous allons développer, et contient uniquement les voisins initiaux du site considéré. D'autres informations globales sont disponibles, l'utilisateur est libre de les utiliser à son gré.

Le type *ensemble* est à noter ; il est associé à une librairie de gestion d'ensembles (fonctions Ens_...) ce qui permet de simplifier nombre d'opérations classiques en algorithmique distribuée. Citons parmi celles-ci, Ens_alea(set) qui effectue un tirage aléatoire d'un élément de set ; Ens_appartient(elt,set) qui teste l'appartenance de l'élément elt dans set, Ens_ajoute, Ens_ote ajoute et supprime respectivement un élément à un ensemble. Autre facilité fournie par la libraire de gestion des ensembles : la fonction Ens_appli(f,set) qui applique la fonction f

successivement à tous les éléments de *set*. En particulier, si l'ensemble *set* est vide, aucune action n'est effectuée. Nous apprécions particulièrement la souplesse de cette dernière fonction qui permet une écriture synthétique et claire de nombreuses applications telle que la diffusion, l'affichage d'ensemble, etc.

4.3.2.c Librairies

En plus de la gestion des émissions/réception de messages, nous avons créé des librairies spécialisées.

Une librairie de gestion des fautes (pannes franches) de lignes est disponible, dans laquelle nous faisons l'amalgame sémantique entre faute de ligne et message signalant cette faute, lequel message est géré comme tout autre message. L'utilisateur cherchant à rendre son algorithme résistant à ce type de panne pourra utiliser cette librairie tout simplement par l'utilisation de paramètres particuliers : la fréquence des pannes et leur nombre ou bien la liste complète des pannes devant survenir. L'amalgame, fautes de ligne et message avec mot-clé spécial, est possible dès que l'on se représente le réseau point-à-point sous la forme d'un réseau physique réel : en effet, un message est l'altération (modulation de fréquence ou modulation d'amplitude) d'une «porteuse», ainsi l'absence de cette porteuse permet de détecter la panne de ligne. L'information «la ligne l est en panne» est traitée après les messages déjà reçus, d'où l'équivalence proposée. La librairie de gestion des fautes de lignes traite aussi l'ajout de lignes au réseau avec la même technique : l'émission d'un message spécial aux deux extrémités de la ligne ajoutée.

Une bibliothèque de réseaux standards est fournie, comprenant l'anneau, la grille, le tore et d'autres structures plus «exotiques»; ce qui n'empêche pas l'utilisateur de pouvoir éventuellement définir son propre réseau...

4.4 Simulations et évaluations d'algorithmes distribués

Nous avons implanté et testé les algorithmes de E. Korach *et al.* [KoKM90], R. G. Gallager *et al.* [GaHS83], I. Lavallée et C. Lavault [LaLa89a], F. Butelle [Bute93]. Pour les juger sur un même plan d'égalité , nous avons choisi de supposer que tous les sites démarrent l'algorithme en même temps et que la topologie est inconnue par les sites. Ce faisant nous les jugeons dans le pire des cas possibles, plus grand est le nombre d'initiateurs plus nombreux sont les conflits.

Des algorithmes de routage ont aussi fait l'objet d'une étude particulière :

versions distribuées de l'algorithme de Dijkstra [Dijk59] et celui de Bellman – Ford [AwBG90], ainsi que l'algorithmes de A. Segall [Sega83]. Ces résultats ont été obtenus en collaboration avec Alain Bui (voir [Bui94]), nous en donnons quelques résultats en annexe.

Nous présentons ici les résultats les plus importants obtenus par la simulation de ces algorithmes, les codes complets pour le simulateur sont présentés en annexe.

4.4.1 Étude d'algorithmes de construction d'AC

Nous allons comparer les algorithmes de : R. G. Gallager *et al.* [GaHS83], I. Lavallée et C. Lavault [LaLa89a], E. Korach *et al.* [KoKM90] avec notre algorithme [Bute93]. L'algorithme de [GaHS83] construit en fait un arbre couvrant de poids total minimum, il figure ici à titre de pionnier de la construction d'arbre couvrant. D'autres algorithmes existent dans la littérature, nous avons étudiés ceux-ci parce qu'ils sont soit très connus soit représentatifs de certaines techniques.

Nous allons considérer les topologies suivantes du graphe associé le plus dense au moins dense : le graphe complet, deux types de réseaux aléatoires, le Tore 2D et l'anneau. Les réseaux aléatoires que nous allons considérer au un degré borné, le premier, au sens de la densité, à un degré compris entre 27 et 30 ; le second a un degré compris entre 6 et 9. Nous obtenons ainsi un échantillon représentatif des topologies possibles, du degré 2 au degré $n-1$ en passant par les degrés 4, 6-9 et 27-30.

Chaque courbe qui va suivre exprime le nombre de messages échangés suivant le nombre de nœuds du graphe sur lequel ces algorithmes ont été simulés, les tailles des messages utilisés par ces algorithmes étant très comparables, une étude en fonction de la quantité d'information échangé n'est pas nécessaire ici. Chaque point représente la moyenne de dix exécutions sur la machine Sequent B8000 dont 9 processeurs sont utilisables par l'utilisateur. Des résultats similaires sont obtenus en simulation séquentielle et sur la KSR1.

Tous les algorithmes requièrent $2m$ messages au moins car il a déjà été prouvé que l'on ne peut faire mieux que ces $2m$ messages (voir le paragraphe 3.1.4 sur l'optimalité des algorithmes distribués de construction d'Arbre Couvrant). Les algorithmes à gestion de phase ne se distinguent en fait que sur un facteur multiplicatif de $n\lg n$, l'algorithme de [GaHS83] nécessite, dans le pire des cas, $2m + 5n\lg n + O(n))$ et l'analyse de l'algorithme de [KoKM90] donne un maximum de $2m + 3\lg n + O(n)$. A notre connaissance, ce dernier résultat le place comme étant le meilleur algorithme de construction répartie d'arbre couvrant en algorithmique distribuée asynchrone.

L'analyse du pire des cas des deux autres algorithmes ([Bute93, LaLa89a]) utilise des exemples « pathologiques » où le nombre de messages est de l'ordre de n^2. Nous verrons que le comportement en moyenne est assez différent.

Dans cette optique, le nombre de messages représenté sur chaque courbe ne prend pas en compte ces $2m$ messages. Nous avons ajouté, représenté par une ligne continue, une courbe « repère » expérimentale : $2n\lg n$. Nous verrons que cette valeur exprime mieux le comportement moyen de la plupart de ces algorithmes.

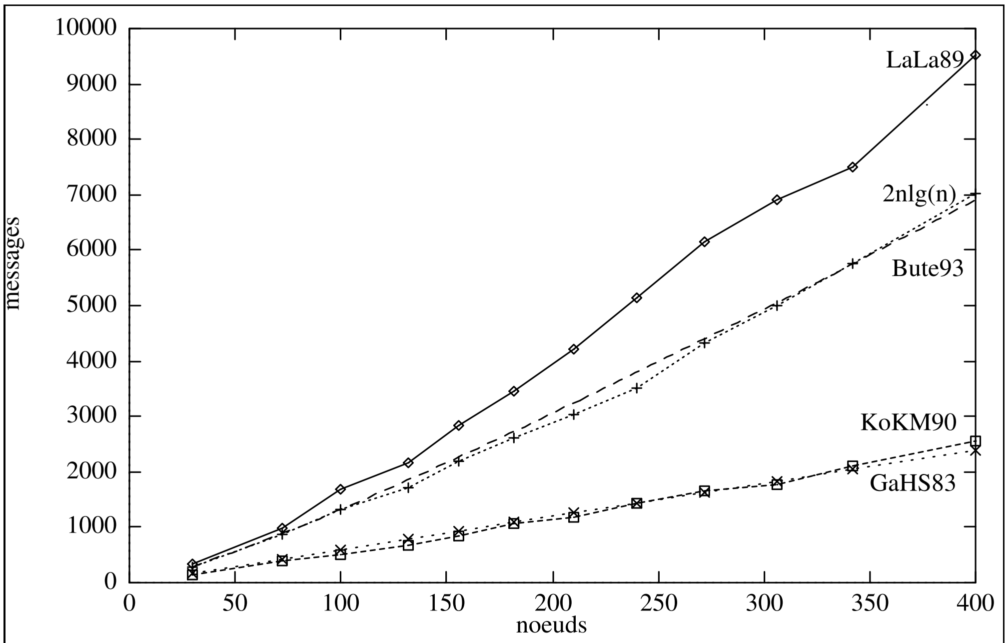

Figure 4.3 : Simulations sur un Anneau Alterné

Dans le cas des anneaux alternés, (voir les figures 4.3 et 2.1), les algorithmes n'utilisant pas la notion de phase [LaLa89a, Bute93] sont conduits à des mises à jours nombreuses qui induisent un nombre de messages relativement important par rapport aux deux autres. Nous avions déjà montré que ces configurations (diamètre proche de n) entraînaient des comportements gourmands en message de part les multiples mises à jour nécessaires.

Pour obtenir la courbe concernant la simulation sur un tore de dimension 2, nous avons en fait estimé que les cotés pouvaient différer d'au plus 1. Ce qui nous a permit d'avoir suffisamment de points pour tracer la courbe de la figure 4.4. Cette courbe montre déjà la grande stabilité des algorithmes à gestion de phase. Les autres ayant des comportement beaucoup moins réguliers, nous avons pu constater des écarts-types très importants variant suivant la taille du réseau. Ils sont souvent supérieurs à 10% du nombre de messages pour [LaLa89a] et sont aussi très importants mais restent inférieurs à 10% pour l'algorithme de [Bute93].

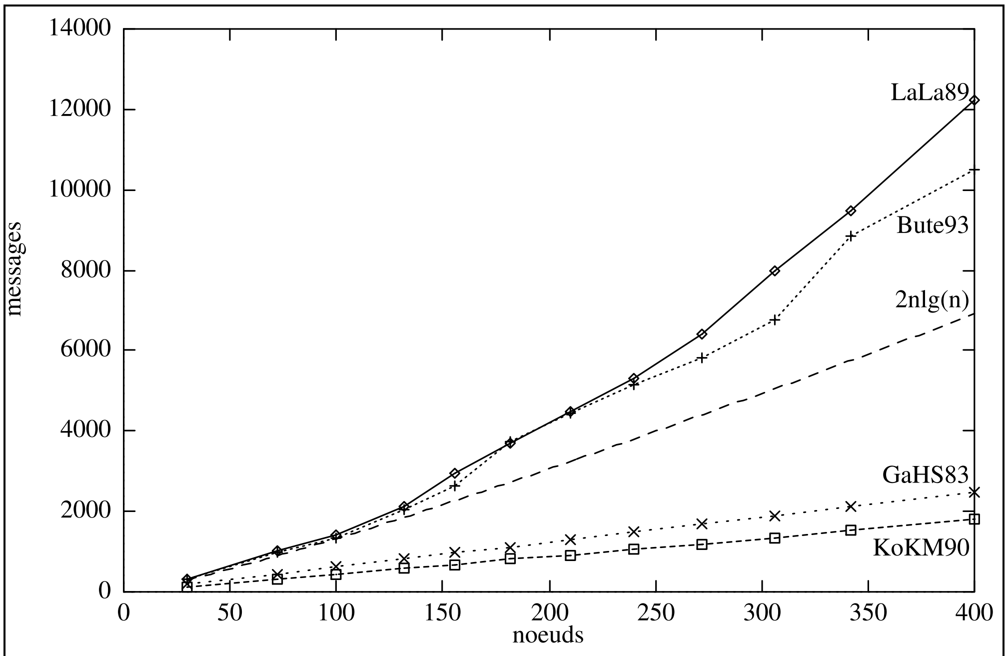

Figure 4.4 : Simulations sur un Tore(2,$n/2$)

Ces deux dernières remarques mettent en évidence le côté imprévisible de ces algorithmes. Ne pas pouvoir prévoir le comportement d'un algorithme correspond à un plus grand potentiel de résistance aux pannes, et on le voit, à un nombre de messages plus important.

Pour exprimer le concept de comportement moyen d'un algorithme distribué fonctionnant sur des graphes quelconques, il est nécessaire de préciser ce que représente un graphe moyen. Personne n'a réellement défini ce qu'est un tel graphe, mais l'on peut figurer un certain comportement moyen par le biais de graphes aléatoires ; nous en étudions de deux types suivant leurs degrés. Puisque nous avons déjà observés les comportements de ces algorithmes sur des graphes de degré inférieur ou égal à 4, il nous a apparu intéressant de considérer des graphes de degrés légèrement supérieur (degré compris entre 6 et 9) et des graphes beaucoup plus denses, de degré compris entre 27 et 30. Ces simulations sont présentés par les figures 4.5 et 4.6.

Les courbes résultant de la simulation sur graphe complet, figures 4.7 et 4.8 montrent combien l'hypothèse d'identités distinctes est coûteuse. Les autres algorithmes confortent leurs positions déjà éprouvée dans les réseaux aléatoires précédents.

Nous avons aussi voulu mesurer et comparer les tailles des buffers nécessaires

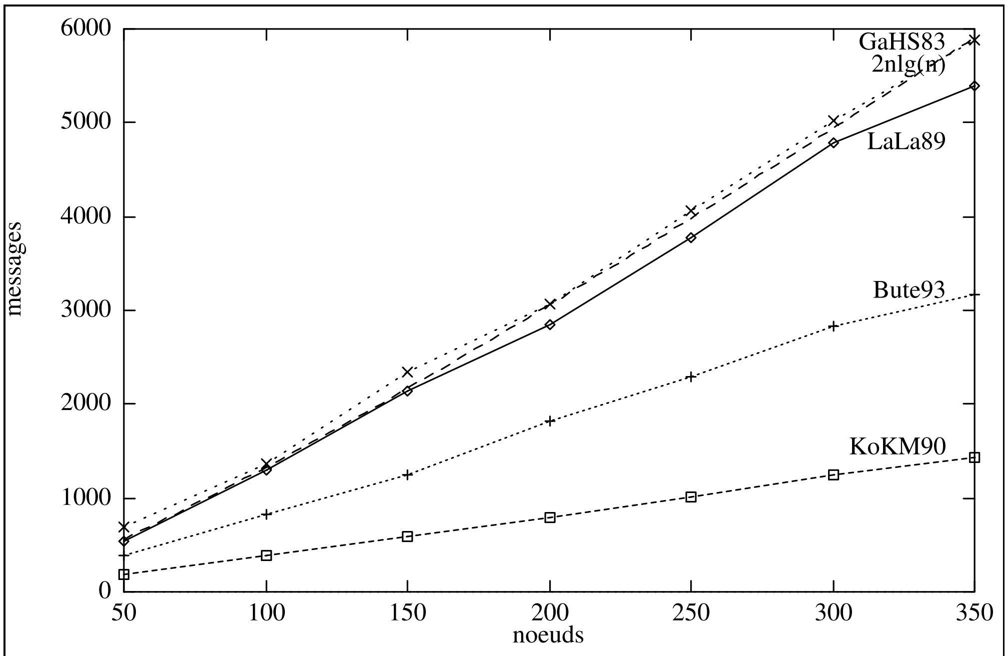

Figure 4.5 : Simulations sur un réseau aléatoire ($6 \leq \delta \leq 9$)

pour chaque algorithme. Nous les avons comparés sur les graphes pour lesquels ils ont *a priori* besoin des plus grandes quantité de mémoire : les graphes complets. En effet, de façon évidente le degré du réseau joue un rôle primordial pour cette taille. Pour qu'un site EGO reçoive un message de ses $n-1$ voisins cela signifie, pour chaque voisin, de choisir tout particulièrement x comme cible. Autrement dit, la probabilité qu'un tel événement survienne est très faible d'où l'intérêt d'une étude en moyenne. Sur trente exécutions, nous avons retenu le maximum du nombre de messages en attente de traitement par site dans la simulation sur graphe complet de 100, 200 et 300 nœuds. Les résultats obtenus figurent dans le tableau suivant :

n	Bute93	GaHS83	KoKM90	LaLa89
100	9	112	11	21
200	11	241	19	35
300	11	369	27	41

Ce tableau nous montre une très forte utilisation des tampons pour [GaHS83] qui résulte du protocole de retardement de traitement de certains messages en les « renfilant » dans les tampons. Cette technique provoque un déséquencement dans l'ordre de traitement des messages ce qui provoque des phénomèmes d'attente.

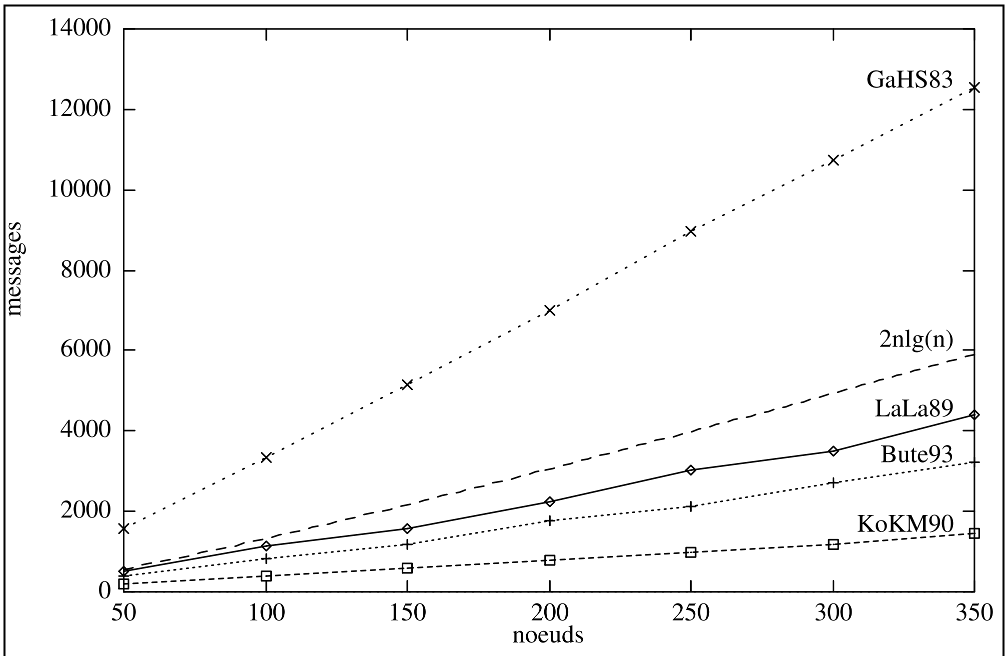

Figure 4.6 : Simulations sur un réseau aléatoire ($27 \leq \delta \leq 30$)

Les autres algorithmes sont relativement plus « sobres » puisque les résultats sont très nettement inférieurs au degré : $n-1$. De très grandes variations ont pu être révélées sur l'algorithme de [KoKM90], les maxima étant atteints qu'assez rarement (environ une fois sur 30, les autres résultats étant jusqu'à deux fois moins importants). Ces maximas figurent le fait, qu'à un instant donné, un comportement plus « parallèle » de l'exécution de l'algorithme a eu lieu. Au vu de la technique utilisée, ce parallélisme est immédiatement limité par une mise en attente de tous les jetons arrivant au profit d'un ou deux qui feront éventuellement un long parcours en profondeur dans le réseau. Cette situation est alors favorable pour le nombre de messages échangés.

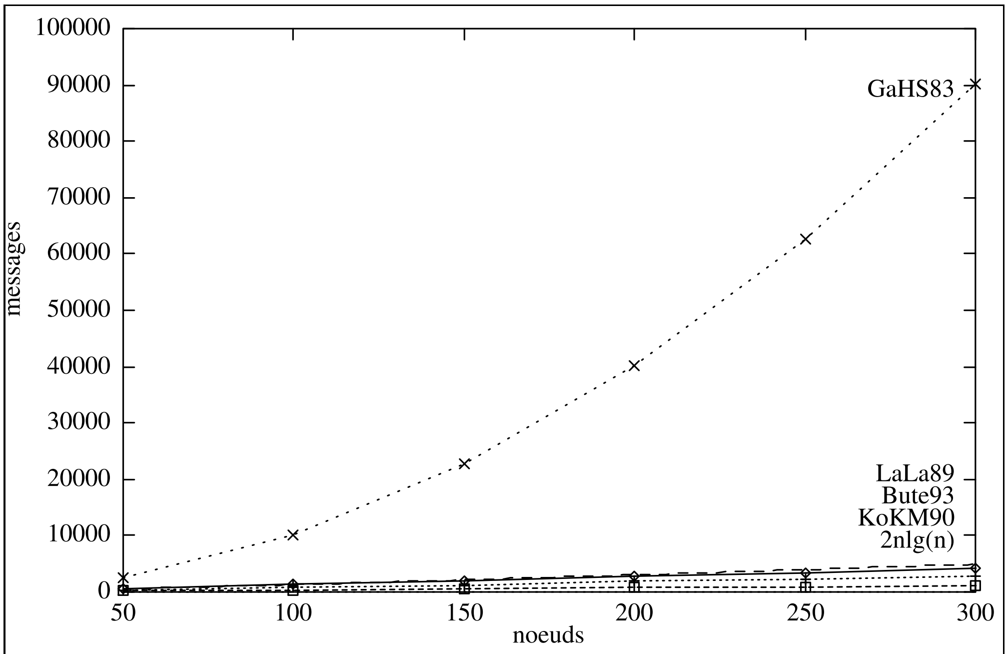

Figure 4.7 : Simulations sur un graphe complet

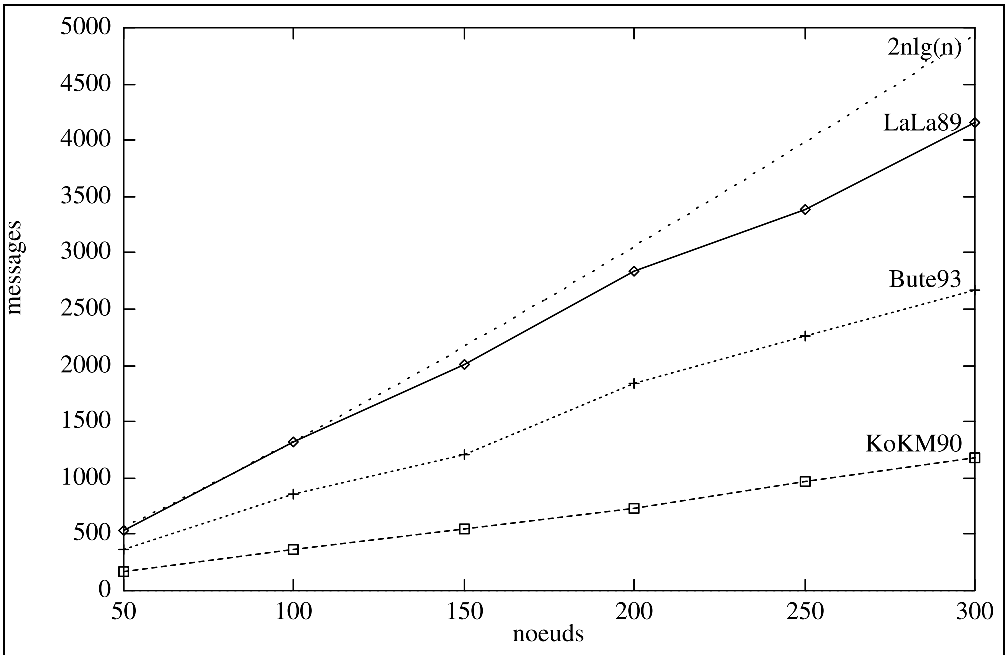

Figure 4.8 : 4.7 sans GaHS83

4.5 Conclusion du chapitre 4

Nous avons présenté dans ce chapitre une étude pratique de plusieurs algorithmes distribués ainsi que le simulateur qui a permis d'obtenir ces résultats sur des topologies de réseau diverses.

Malgré l'arrivée de machines réellement distribués comme nous l'a montré le chapitre 2, un simulateur est encore un outil nécessaire au programmeur. En effet, de notre propre expérience, nous avons pu constater les difficultés particulières que représente chaque machine parallèle (ou distribuée), chacune a ses caractéristiques, ses possibilités et ses défauts. Développer un nouveau code pour chaque nouvelle machine rencontrée, et tenter d'obtenir des résultats optimaux par rapport à la machine, est un travail fastidieux rarement intéressant et rendu difficile par des descriptions techniques disponibles que par morceaux ou de façon très diluée dans des documentations de milliers de pages. Nous avons donc développé ce simulateur aussi à des fins de simplification de l'utilisation de ces machines pour toute personne intéressée par la programmation par échange de message et ayant un minimum de connaissances en langage C. Pour pallier au problème de l'implantation, nous avons établi des librairies indépendantes et réduit le noyau spécifique machine au minimum. Adapter notre code pour une nouvelle machine parallèle ne consiste plus alors qu'à une réécriture de ce noyau, en général une vingtaine de lignes de macros C suffisent.

Notre simulateur est aussi un évaluateur, nous avons pu mettre en évidence des comportements catastrophiques pour certains algorithmes, telle l'apparition de phénomènes d'engorgement local pour des algorithmes de routage. La partie évaluation tend à une évolution plus théorique grâce à notre coopération avec M. Bui : l'analyse semi-automatique du comportement de l'algorithme étudié via l'utilisation de chaînes de Markov. Cette analyse pourra permettre de positionner certains paramètres de l'algorithme en vue de l'optimisation de celui-ci (voir [Bui89, Bui90a]).

Conclusion

Nous avons présenté dans cete thèse, une étude constructive de la problématique de l'algorithmique distribuée dans ce qu'elle a de plus original mais aussi de plus difficile à appréhender : l'algorithmique asynchrone.

Le concept d'algorithmique distribuée asynchrone conduit à la recherche de structures de contrôle sur la totalité du réseau. Nous nous sommes efforcés de définir un modèle précis et simple très peu exigeant en matière de connaissances globales : pas d'horloge globale ni de mémoire partagée.

La taille d'un réseau et l'arbitraire de sa topologie conduisent à écarter des techniques de type routage point à point et anneau virtuel pour l'établissement de structure de contrôle. Nous avons donc montré des algorithmes de construction d'arbre couvrant ayant chacun des particularités intéressantes.

De nombreux problèmes classiques de l'algorithmique distribuée tels que l'élection, la recherche d'extremum et la terminaison distribuée sont réductibles à la construction d'un Arbre Couvrant comme nous l'avons montré dans le chapitre 3. Cette élection est, dans la littérature, généralement prédeterminée par le réseau en fonction de la distribution des identités des sites. Nous avons présenté des algorithmes qui élisent un site au hasard, contrairement à la majorité des algorithmes qui élisent un extremum, ce qui garantie une certaine résistance aux pannes et aux tentatives d'espionnage.

Une structure de contrôle distribuée se doit d'être efficace, nous avons présenté deux contraintes de construction : le poids total minimal qui traduit plutôt une recherche économique et le diamètre minimal. De façon évidente une structure de contrôle de diamètre (valué) minimum traduit la recherche d'efficacité aussi bien en messages qu'en temps lorsque les liens de communications sont valués par leurs délais de transmission.

Construire un arbre couvrant de diamètre minimal est un problème nouveau en algorithmique distribuée, nous avons proposé une méthode simple, utilisant des algorithmes de parcours connus. Cette méthode permet de décrire différents algorithmes distribués de construction d'Arbre Couvrant de Diamètre Minimal fonctionnant sur des réseaux avec ou sans pannes.

Nous avons aussi présenté un simulateur qui de par sa réalisation nous a apporté une expérience certaine dans la programmation de machines parallèles. Ce simulateur nous a aussi permis de mettre en évidence certains comportements à occurrences rares dans les algorithmes présentés ainsi que des comportements en moyenne que l'analyse théorique n'a pas révélés. Nous avons explicités ces comportements et apporté des solutions simples. Une fois codés dans un langage de description simple et intuitif, les algorithmes ont été simulés sur des graphes divers et nous ont permis d'établir des distinctions entre algorithmes de construction d'arbre couvrant établissant ainsi un classement suivant différents critères.

Annexe A

Symboles standards

$\mathbb{R}$ Ensemble des nombres réels.

$\mathbb{N}$ Ensemble des entiers naturels.

$\mathbb{Z}$ Ensemble des entiers relatifs.

$\emptyset$ Ensemble vide.

$|\boldsymbol{V}|$ Cardinal de l'ensemble V.

$\{\boldsymbol{x}/\boldsymbol{x}$ **tel que** $\ldots\}$ Ensemble des x tels que...

$\boldsymbol{a} \in \boldsymbol{A}$ a appartient à l'ensemble A.

$\exists \boldsymbol{a} \in \boldsymbol{A}$ Il existe a appartenant à A tel que...

$\forall \boldsymbol{a} \in \boldsymbol{A}$ Quel que soit l'élément a de A..., *ou* pour tous les éléments a de A...

$\boldsymbol{P} \wedge \boldsymbol{Q}$ P *et logique* Q où P et Q sont des propositions logiques.

$\boldsymbol{P} \vee \boldsymbol{Q}$ P *ou logique* Q.

$\neg \boldsymbol{P}$ *non logique* de P.

$\boldsymbol{A} \bigcup \boldsymbol{B}$ Union de A et B.

$\boldsymbol{A} \bigcap \boldsymbol{B}$ Intersection de A et B.

$\boldsymbol{A} - \boldsymbol{B}$ A moins B (éléments de A qui n'appartiennent pas à B).

$\boldsymbol{A \subset B}$ A inclus dans B.

$\boldsymbol{\Gamma(a)}$ Image de l'élément a dans la correspondance Γ.

$\boldsymbol{\Gamma(A)}$ Image de l'ensemble A dans la correspondance Γ, égal à : $\bigcup_{a \in A} \Gamma(a)$.

$\boldsymbol{\Gamma^{-1}(A)}$ Correspondance inverse de la correspondance Γ, définie par $\Gamma^{-1}(y) = \{x / y \in \Gamma(x)\}$.

ln$\boldsymbol{p}$ logarithme Népérien de p.

lg$\boldsymbol{p}$ logarithme à base 2 de p.

$\boldsymbol{\log_b p}$ logarithme à base b de p. Nous utiliserons aussi la forme log pour les ordres de grandeur des complexités des algorithmes puisqu'alors la base n'est pas significative.

$\boldsymbol{\lceil p \rceil}$ partie entière par excès de p.

$\boldsymbol{\lfloor p \rfloor}$ partie entière par défaut de p.

Annexe B

Code de quelques algorithmes

B.1 Algorithme de [Bute93]

```
/* Algorithme d'arbre couvrant avec racine non pre-determinee.  Plutot
qu'une gestion par etats, nous avons choisi d'affiner la notion de
candidat et la notion d'actif.*/

VAR                                                  /* Var. locales a tout site */
  Octet
    req,open,free,free_mesg,open_mesg;
  Ensemble
    fils;
  Octet
    candidate[PIDMAX],active[PIDMAX];
  Pidtype
    id_mesg,last,pred,idcomp;
FIN

#define b_ FAUX                                      /* booleen presque indefini !*/
#define PRIOR 2                                      /* Candidats prioritaires */
#define EVITE 3                                      /* Candidats a eviter si possible */

MCL                                                  /* Mots-cles des messages */
    end,conn,ok,nok,cousin,newroot,updt,token;
FIN

/*————————————————————————————*/
void *mon_printf(iter)                               /* Utilisee par Ens_appli */
Mot iter;
{
```

```
   printf("%4d",iter);
   if (candidate[iter])
      printf("C");
   else if (active[iter])
      printf("A");
}
FIN

AFFICHE                                        /* Pour les traces d'execution */
   printf("%d:",Mypid);
   Ens_appli(mon_printf,fils);
   printf(",Id=%u,p=%u,l=%u,",idcomp,pred,last);
   if (free) printf("F");
   if (open) printf("O");
   printf("\n");
FIN

/*————————————————————————————*/
void envoi(lui,mcl,id,bool1,bool2)
Mot lui,id;
Octet bool1,bool2,mcl;
{
   char inter[8],c1,c2;

   c1= (bool1 ? 'V' : 'F');
   c2= (bool2 ? 'V' : 'F');
   sprintf(inter,"%c%c%u",c1,c2,id);
   Envoi(lui,mcl,inter);
}
FIN

/*————————————————————————————*/
defile()
{
   unsigned i;

   if(Defile())
   {
       open_mesg= Mesg[1]=='V';
       free_mesg= Mesg[0]=='V';
       sscanf(Mesg+2,"%u",&i);
       id_mesg=(Pidtype)i;
       return(VRAI);
   }
   else
       return(FAUX);                           /* Pas de message en attente */
}
FIN
```

```
/*——————————————————————————————*/
void UPDT(x,act,cand)        /* Mise a jour de l'etat de x suivant act et cand*/
Mot x;
Octet act,cand;
{
  int i;
  Octet res_a,res_c;
  active[x]= act; candidate[x]=cand;

  if (cand)
  {
      open=VRAI;free=VRAI;
  }
  else
  {
      for (i=1,res_a=FAUX,res_c=FAUX;i≤Nbpid && !res_c;i++)
      {
          if(candidate[i])
          {
              res_c=VRAI; res_a = VRAI;
          }
          else if(active[i])
              res_a = VRAI;
          if(candidate[i] && !active[i])
              erreur("pb : candidate/active\n");
      }
      open = res_c; free= res_a;
  }
}
FIN                                              /* UPDT(x,act,cand) */
/*——————————————————————————————*/
Mot Select()                                     /* Selection d'1 candidat */
{
  int i,j,nbposs;
  Mot possible[PIDMAX];

  nbposs=0;                          /* Candidats prioritaires et non-fils */
  for (i=1;i≤Nbpid;i++)
      if (candidate[i]==PRIOR && !Ens_appartient(i,fils))
          possible[nbposs++]=i;

  if (nbposs==0)                                 /* Candidats non-fils */
      for(i=1;i≤Nbpid;i++)
          if (candidate[i] && !Ens_appartient(i,fils))
```

```
                possible[nbposs++]=i;
  if (nbposs==0)                        /* Candidats qui ne sont pas a eviter */
      for(i=1;i≤Nbpid;i++)
          if (candidate[i] && candidate[i]≠ EVITE)
              possible[nbposs++]=i;

  if (nbposs==0)                            /* Sinon, on prend ce qu'il y a ! */
      for(i=1;i≤Nbpid;i++)
          if (candidate[i])
          {
              possible[nbposs++]=i;
              candidate[i]=1;        /* On remet les priorites au meme niveau*/
          }
  if (nbposs==0)
    erreur("Pas de candidat a Select.");

  j=possible[random()%nbposs];       /* Cas moyen. Pire des cas : possible[0] */
  if (Ens_appartient(j,fils))
      candidate[j]=1;
  return(j);
}
FIN                                                              /* Select() */
/*------------------------------------------------------*/
Octet Or(t)                               /* Ou logique des elts du tableau t */
Octet *t;
{
    int i=1;

    while(i≤Nbpid && t[i]==0) i++;
    return(i≤Nbpid);
}
FIN

void *init(iter)                                     /*Utilisee par Ens_appli*/
Mot iter;
{
    candidate[iter]=1;active[iter]=1;
}
FIN

void *env_newroot(iter)                              /*Utilisee par Ens_appli*/
Mot iter;
{
    envoi(iter,newroot,idcomp,b_,b_);
}
FIN
```

```
void *env_end(iter)                                    /*Utilisee par Ens_appli*/
Mot iter;
{
    envoi(iter,end,idcomp,b_,b_);
}
FIN

CALCUL                                                       /* LE programme */
  int i,j;
  Octet b1,b2;

  1:                                                        /* Initialisation */
    pred= 0; req=FAUX; open=VRAI;free=VRAI;
    last = 0;
    idcomp=Mypid;
    Ens_appli(init,Voisins);Ens_nul(fils);
    Va_en 2;

  2:                                         /* Routine principale de l'algo*/
    if (pred==0 && !req && !free)
          Va_en 22;
    else if (pred==0 && !req && open)
          Va_en 21;
    else if (defile())
          Va_en 3;
    break;

  21:                                /* Tentative de connection  ou dep. jeton */
    j=Select();
    if (Ens_appartient(j,fils))
    {                          /* deplacement de racine par emission de jeton */
        last=pred=j;Ens_ote(j,fils);
        UPDT(j,FAUX,FAUX);
        envoi(j,token,idcomp,free,open);
    }
    else                                            /* Tentative de connection */
    {
        envoi(j,conn,idcomp,b_,b_);
        req=VRAI;
    }
    Snapshot=VRAI;
    Va_en 2;
    break;

  22:                                                          /* Terminaison */
    printf("Elu:%d,\n",Mypid);
    Ens_appli(env_end,fils);
    THE END;
```

```
    break;
  3:                                          /* reception de message */
    BRANCH_ON Mcl;
    break;
  ON_MESG end:                                    /* reception de end */
    Ens_appli(env_end,fils);
    THE END;
    break;
  ON_MESG conn :                                 /* reception de conn */
    if (idcomp > id_mesg)
    {                                                       /* Fusion */
        envoi(Org,ok,idcomp,b_,b_);
        Ens_ajoute(Org,fils);
        candidate[Org]=1;
        if (pred≠0 && !open)
            envoi(pred,updt,idcomp,free,VRAI);
        open=VRAI;
    }
    else if (idcomp== id_mesg)
    {                                                /* Refus : cousin */
        envoi(Org,cousin,idcomp,b_,b_);
        candidate[Org]=0;active[Org]=0;
        b1=Or(active);b2=Or(candidate);
        if (pred≠0 && (free ≠ b1) || open≠ b2)
            envoi(pred,updt,idcomp,b1,b2);
        free=b1;open= b2;
    }
    else                                            /*id_mesg>idcomp*/
    {                                              /* Refus : plus gros */
        envoi(Org,nok,idcomp,b_,b_);
        candidate[Org]=PRIOR;                   /* candidat prioritaire */
        if (!open && pred≠0)
            envoi(pred,updt,idcomp,free,VRAI);
        open=VRAI;
    }
    Va_en 2;

  ON_MESG ok :                                     /* reception de ok */
    req=FAUX;idcomp=id_mesg;pred=Org;
    UPDT(Org,FAUX,FAUX);
    Ens_appli(env_newroot,fils);
    Va_en 2;

  ON_MESG nok :                                   /* reception de nok */
```

```
    if (!Ens_appartient(Org,fils))
    {
        candidate[Org]=0;
        open=Or(candidate);
    }
    req=FAUX;
    Va_en 2;

  ON_MESG cousin :                                   /* reception de cousin */
    UPDT(Org,FAUX,FAUX);
    req=FAUX;
    Va_en 2;

  ON_MESG newroot :                                  /* reception de newroot */
    idcomp=id_mesg;
    Ens_appli(env_newroot,fils);
    Va_en 2;

  ON_MESG updt :                                     /* reception de updt */
    if (pred ≠Org)
    {
        candidate[Org]=open_mesg;active[Org]=free_mesg;
        if (pred≠0 &&
            (open ≠ Or(candidate) || free ≠ Or(active)))
            envoi(pred,updt,idcomp,Or(active),Or(candidate));
        open = Or(candidate);free=Or(active);
    }
    Va_en 2;

  ON_MESG token :                                    /* reception de token */
    pred=0;Ens_ajoute(Org,fils);last=Org;
    UPDT(Org,free_mesg,open_mesg);
    if (candidate[Org])
      candidate[Org]=EVITE;          /* evite de relancer le jeton sur le dernier
                                     qui la possede */
    Snapshot=VRAI;
    Va_en 2;
FIN
```

B.2 Algorithme de [GaHS83]

```
VAR
  Octet edge_state[PIDMAX],state,state_mesg;
  Mot level,level_mesg,best_edge,test_edge,in_branch,find_count;
```

```
   unsigned int frag,frag_mesg,best_wt;
FIN

#define Find 'f'                                  /* les etats d'un site */
#define Found 'F'
#define Basic 'b'                                 /* les etats d'un lien */
#define Branch 'B'
#define Rejected 'R'
#define NIL 0
#define INFINI (unsigned int)(~0)

MCL                                       /* Les mots-cles des messages */
conn,accept,reject,newroot,init,report,test;
FIN
/*-------------------------------------------------------*/
AFFICHE
   int j;

   printf("%3d,%3d,F=%5d,S=%c,IB=%3d,c=%3d,",
            Mypid,level,frag,state,in_branch,find_count);
   printf("BE=%3d,BW=%5d,TE=%3d,SE=",best_edge,best_wt,test_edge);
   for(j=1;j<=Nbpid;j++)
      if (edge_state[j]==Branch)
            printf("%3d",j);
      else if (edge_state[j]==Rejected)
            printf("%3dR",j);
   printf("\n");
FIN
/*------------------------------------------------------------*/
void envoi(lui,l,f,etat,mcl)
Mot lui;
unsigned int f;
Octet l,etat,mcl;
{
   char inter[20];

   inter[0]=l+'0';
   inter[1]=etat;

   sprintf(inter+2,"%u",f);
   Envoi(lui,mcl,inter);
}
FIN
/*--------------------------------------------------------------*/
Octet defile()
{
```

```
  static char inter[10];

  if(Defile())
  {
      level_mesg=Mesg[0] -'0'; state_mesg=Mesg[1];
      sscanf(Mesg+2,"%u",&frag_mesg);

      return(VRAI);
  }
  else
      return(FAUX);
}
FIN
proc_report()                                    /* Procedure report */
{
    if (find_count==0 && test_edge==NIL)
    {
        state=Found;
        envoi(in_branch,NIL,best_wt,state,report);
    }
    Va_en 2;
}
FIN
proc_test()                                        /* Procedure test */
{
  int i,j;
    for(j=0,i=1;j==0&&i≤Nbpid;i++)
        if (Ens_appartient(i,Voisins) && edge_state[i]==Basic)
            j=i;
    if (j≠0)
    {
        test_edge=j;
        envoi(j,level,frag,state,test);
        Va_en 2;
    }
    else
    {
        test_edge=NIL;
        proc_report();
    }
}
FIN
proc_newroot()                                  /* Procedure newroot */
{
```

```
    if (edge_state[best_edge]==Branch)
        envoi(best_edge,NIL,NIL,state,newroot);
    else
    {
        envoi(best_edge,level,NIL,state,conn);
        edge_state[best_edge]=Branch;
    }
    Va_en 2;
}
FIN

/*--------------------------------------------------*/
CALCUL
int i,j;

1 : for (i=1,j=0;i≤Nbpid;i++)                          /* Initialisations */
         if (Ens_appartient(i,Voisins))
            if (j==0)
            {
                edge_state[i]= Branch;
                j=i;
            }
            else
                edge_state[i]=Basic;

        level=0;state=Found;find_count=0;
        envoi(j,0,NIL,state,conn);
        Va_en 2;

2 :                                           /* Routine principale de l'algo*/
        if (defile())
          BRANCH_ON Mcl;
        break;

ON_MESG conn :                                          /* reception de conn */
        if (level_mesg<level)
        {
          edge_state[Org]=Branch;
          envoi(Org,level,frag,state,init);
          if (state == Find)
            find_count++;
        }
        else if (edge_state[Org] == Basic)
          Renfile();                      /* remet le mesg en fin de file d'attente */
        else
        {
          if (Mypid < Org)                   /* calcul arbitraire du poids du lien */
```

```
            j=(Nbpid+1)∗ Mypid + Org;
        else
            j=(Nbpid+1)∗ Org + Mypid;
        envoi(Org,level+1,j,Find,init);
    }
    Va_en 2;

ON_MESG init :                                          /∗ reception de init ∗/
    level=level_mesg;frag=frag_mesg;state=state_mesg;in_branch=Org;
    best_edge=NIL;best_wt=INFINI;
    for(i=1;i≤Nbpid;i++)
        if (i≠Org && edge_state[i]==Branch)
        {
            envoi(i,level,frag,state,init);
            if (state==Find) find_count++;
        }
    if (state==Find)
        proc_test();
    else
        Va_en 2;
    break;
ON_MESG test :                                          /∗ reception de test ∗/
    if (level_mesg>level)
    {
        Renfile();
        Va_en 2;
    }
    else if (frag_mesg ≠ frag)
    {
        envoi(Org,NIL,NIL,state,accept);
        Va_en 2;
    }
    else
    {
        if (edge_state[Org]==Basic)
            edge_state[Org]=Rejected;
        if (test_edge == Org)
            proc_test();
        else
        {
            envoi(Org,NIL,NIL,state,reject);
            Va_en 2;
        }
    }
    break;
```

```
ON_MESG accept :                                      /* reception de accept*/
        test_edge=NIL;
        if (Mypid < Org)
          j=PIDMAX* Mypid + Org;
        else
          j=PIDMAX* Org + Mypid;
        if (j<best_wt)
        {
          best_wt=j;best_edge=Org;
        }
        proc_report();
        break;
ON_MESG reject :                                      /* reception de reject*/
        if (edge_state[Org]==Basic)
          edge_state[Org]=Rejected;
        proc_test();
        break;
ON_MESG report :                                      /* reception de report*/
        if (Org ≠in_branch)
        {
          find_count--;
          if (frag_mesg < best_wt)
          {
            best_wt = frag_mesg;best_edge=Org;
          }
          proc_report();
        }
        else if (state==Find)
        {
          Renfile();
          Va_en 2;
        }
        else if (frag_mesg>best_wt)
          proc_newroot();
        else if (frag_mesg==best_wt && best_wt==INFINI)
        {
           if (Org==in_branch && Mypid<Org &&
               frag==Mypid*(Nbpid+1)+Org)
             printf("Elu: %d\n",Mypid);
          for (i=1;i≤Nbpid;i++)                 /* ajout pour la terminaison*/
            if (i≠Org && edge_state[i]==Branch)
              envoi(i,NIL,INFINI,state,report);
          THE END;                                        /* C'est fini */
        }
```

```
        break;
ON_MESG newroot :                              /* reception de newroot */
        proc_newroot();
        break;
FIN
```

B.3 Algorithme de [LaLa89a]

/ Algorithme d'arbre couvrant avec racine = Id. max, (c.f. RR 1024). Nous avons decoupe le message nok en deux : nok et noki resp. pour nok externe et nok interne. */*

```
VAR
  char
    req,free,open,lib_mesg,ouv_mesg;
  Pidtype
    idcomp,id_mesg,pred;
  Ensemble
    active,candidate,fils;
FIN

#define LIBRE 'L'                                       /* free */
#define OUVERT 'O'                                      /* open */
#define NON 'N'
#define NIL 0

MCL                                  /* Les mots-cles des messages */
end,conn,ok,oki,nok,cousin,newroot,opening,noki,merge;
FIN

/*------------------------------------------------*/
AFFICHE
  printf("%3d,%3d,P=%3d,Op=",Mypid,idcomp,pred);
  printf(",Et=%c%c",free,open);
  printf(",A=");Ens_aff(active);
  printf(",C=");Ens_aff(candidate);
  printf(",F=");Ens_aff(fils);
  if(req) printf("R");
  printf("\n");
FIN

/*------------------------------------------------*/
void envoi(lui,id,l,o,mcl)
Mot lui,id;
char l,o;
```

```
Keywordtype mcl;
{
    static char inter[8];

    sprintf(inter,"%c%c%u",l,o,id);
    Envoi(lui,mcl,inter);
}
FIN
/*————————————————————————-*/
defile()
{
    char c1,c2;
    unsigned i;
    if (Defile())
    {
        sscanf(Mesg,"%c%c%u",&c1,&c2,&i);
        lib_mesg=c1;ouv_mesg=c2;
        id_mesg=(Pidtype)i;
        return(VRAI);
    }
    else
        return(FAUX);
}
FIN
/*——————————————————————————————*/
void UPDT(x,l,o)
Pidtype x;
char l,o;
{
    int i;
    if (l==LIBRE)
        Ens_ajoute(x,active);
    else
        Ens_ote(x,active);
    if (o==OUVERT)
        Ens_ajoute(x,candidate);
    else
        Ens_ote(x,candidate);
    if (Ens_or(candidate))
    {
        free=LIBRE;open=OUVERT;
    }
    else if(Ens_or(active))
```

```
    {
        free=LIBRE;open=NON;
    }
    else
    {
        free=NON;open=NON;
    }
}
FIN
/*-----------------------------------------------------------*/
Mot Select()                                    /* Selection d'1 candidat */
{
    int i,nbposs;
    Mot possible[PIDMAX];

    nbposs=0;
    if (nbposs==0)
        for(i=1;i≤Nbpid;i++)                    /* Pire des cas : id. inf. */
            if (Ens_appartient(i,candidate) && !Ens_appartient(i,fils))
                possible[nbposs++]=i;

    if (nbposs==0)
        if((i=Ens_alea(candidate))==0)
            erreur("Pas de candidat a Select.");
        else
            return(i);
    else
        return(possible[random() % nbposs]);    /* Cas moyen. Pire des cas:
                                                 * possible[0] */
}
FIN

void *env_end(iter)             /* Utilisee par Ens_appli pour terminaison */
Mot iter;
{
envoi(iter,idcomp,free,open,end);
}
FIN

void *env_newroot(iter)                         /* Utilisee par Ens_appli */
Mot iter;
{
envoi(iter,idcomp,free,open,newroot);
}
FIN

/*-----------------------------------------------------------*/
CALCUL
```

```
1 :                                              /* Initialisation */
    pred= NIL; req=FAUX; open=OUVERT;free=LIBRE;
    idcomp=Mypid;Ens_nul(fils);
    Ens_copy(Voisins,candidate);
    Ens_copy(Voisins,active);
    Va_en 2;
2 :                                   /* Routine principale de l'algo*/
    if (idcomp==Mypid && !req && open==OUVERT)
       Va_en 21;
    else if (idcomp==Mypid && !req && free==NON)
       Va_en 22;
    else if (defile())
       BRANCH_ON Mcl;
    break;
21 :                                       /* Tentative de connection */
    envoi(Select(),idcomp,free,open,conn);
    req=VRAI;
    Snapshot=VRAI;
    Va_en 2;
22 :                                                  /* C'est fini */
    Ens_appli(env_end,fils);
    THE END;
ON_MESG end :                                    /* reception de end */
    Ens_appli(env_end,fils);
    THE END;
ON_MESG conn :                                  /* reception de conn */
    if (Org == pred)
      if (open==OUVERT)
          envoi(Select(),idcomp,free,open,conn);
      else
      {
          envoi(pred,idcomp,free,open,noki);
      }
    else if (id_mesg< idcomp)
    {
        envoi(Org,idcomp,free,open,ok);Ens_ajoute(Org,fils);
        UPDT(Org,LIBRE,NON);
    }
    else if (id_mesg==idcomp)
    {
        envoi(Org,idcomp,free,open,cousin);
        UPDT(Org,NON,NON);
        if (free==NON && Mypid≠idcomp)                    /* Ajout */
```

```
            envoi(pred,idcomp,NON,NON,merge);
    }
    else                                                    /*id_mesg>idcomp*/
    {
        envoi(Org,idcomp,free,open,nok);
        Ens_ajoute(Org,candidate);
        if (open≠OUVERT && Mypid≠idcomp)
            envoi(pred,idcomp,free,OUVERT,opening);
        open=OUVERT; Snapshot=VRAI;
    }
    Va_en 2;
ON_MESG oki :                                       /* version interne du ok */
    req=FAUX;
    if (Mypid==idcomp)
    {
        Ens_ote(Org,fils);
        idcomp=id_mesg; pred=Org;
        UPDT(Org,NON,NON);
        Ens_appli(env_newroot,fils);
        envoi(Org,idcomp,free,open,merge);
    }
    else
    {
        Ens_ote(Org,fils);
        Ens_ajoute(pred,candidate);
        Ens_ajoute(pred,active);
        UPDT(Org,NON,NON);
        idcomp=id_mesg;
        envoi(pred,id_mesg,free,open,oki);
        Ens_appli(env_newroot,fils);
        Ens_ajoute(pred,fils);pred=Org;
    }
    Va_en 2;
ON_MESG ok :                                            /* reception de ok */
    req=FAUX;
    Ens_ote(Org,fils);                  /* On prepare le retournement de chemin */
    UPDT(Org,NON,NON);
    idcomp=id_mesg;                        /* nouvelle identite de composante */

    if (Mypid==idcomp)               /* Mypid etait la racine (donc pas de pred) */
        envoi(Org,idcomp,free,open,merge);
    else
    {
        Ens_ajoute(pred,candidate);
```

```
        Ens_ajoute(pred,active);
        envoi(pred,id_mesg,free,open,oki);
        Ens_ajoute(pred,fils);
    }
    pred=Org;                                   /* Nouvelle orientation */
    Ens_appli(env_newroot,fils);   /* M.a.j. des id. de composante des fils */
    Va_en 2;

ON_MESG noki :                              /* version interne du nok */
    req=FAUX;
    UPDT(Org,lib_mesg,ouv_mesg);
    if (Mypid ≠ idcomp)
      if (open==OUVERT)
          envoi(Select(),idcomp,free,open,conn);
      else
          envoi(pred,idcomp,free,open,noki);
    Va_en 2;

ON_MESG nok :                                    /* reception de nok */
    req=FAUX;
    if (!Ens_appartient(Org,fils))
    {
        UPDT(Org,LIBRE,NON);
    }
    if (Mypid ≠ idcomp)
       if (open==OUVERT)
           envoi(Select(),idcomp,free,open,conn);
       else
           envoi(pred,idcomp,free,open,noki);
    Va_en 2;

ON_MESG cousin :                              /* reception de cousin */
    UPDT(Org,NON,NON);
    req=FAUX;
    if (Mypid≠idcomp)
       if (open==OUVERT)
           envoi(Select(),idcomp,free,open,conn);
       else if (pred≠NIL)
           envoi(pred,idcomp,free,open,noki);
    Va_en 2;

ON_MESG newroot :                            /* reception de newroot */
    idcomp=id_mesg;
    Ens_appli(env_newroot,fils);
    Va_en 2;

ON_MESG merge:                                 /* reception de merge */
    UPDT(Org,lib_mesg,ouv_mesg);
```

```
    if (idcomp ≠ Mypid)
        envoi(pred,idcomp,free,open,merge);
    Va_en 2;
ON_MESG opening :                          /* reception de opening */
    if (pred≠0)
       if (open==NON)
            envoi(pred,idcomp,LIBRE,OUVERT,opening);
    UPDT(Org,LIBRE,OUVERT);
    Va_en 2;
FIN
```

B.4 Algorithme de [KoKM90]

```
/* Implementation simplifiee : FIFO respectee pour les tampons; tous
demarrent l'algo. */
#define FORMAT_MSG "%u,%d,%u,%u"
VAR
  int
    phase,phase_mesg,candidat,chased;
  Octet
    sleep,termine_porteur,mocl_mesg,marquage_mcl;
  Pidtype
    last,pere,dom,dom_mesg,init_mesg,marquage_dom;
  Ensemble
   candidats,non_cousins;
FIN
MCL
annex,chase,end,dfs,dfs_back,cousin;
FIN
AFFICHE
  printf("%d-%d(%d):",Mypid,phase,dom);
  if (candidat ≠ -1) printf("[%d]",candidat);
  if (last) printf("l=%d,m=%d",last,marquage_dom);
  if (chased ≠ -1) printf(",ch");
  printf("; non cous.:");
  Ens_aff(non_cousins);
  printf("\n");
FIN
/*——————————————————————————*/
/* Ici l'algo "porteur" */
```

```
envoi(lui,mocl1,mocl2,p,d,i)
Pidtype lui;
Keywordtype mocl1,mocl2;
int p;
Pidtype d,i;
{
    static char chaine[60];

    sprintf(chaine,FORMAT_MSG,mocl2,p,d,i);
    Envoi(lui,mocl1,chaine);
}
FIN

void *env_end(iter)                              /* Utilisee par Ens_appli */
Pidtype iter;
{
    envoi(iter,end,end,phase,dom,dom);
}
FIN

void init(mcl,p,a)                  /* init. un nouvel algo. porteur de type DFS */
Keywordtype mcl;
int p;
unsigned int a;
{
    pere=0;termine_porteur=FAUX;
    marquage_dom=Mypid;marquage_mcl=mcl;
    if (mcl≠ chase)
    {
        Ens_copy(non_cousins,candidats);
        last=Ens_alea(candidats);
    }
    else     /* Si il s'agit d'une "chasse" : on suit le chemin donne par last */
        Ens_copy(Voisins,candidats);

    envoi(last,mcl,dfs,p,a,Mypid);
    Ens_ote(last,candidats);
}
FIN

void cont(mcl,p,a)           /* Continue l'algo porteur avec un nouveau message */
Keywordtype mcl;
int p;
Pidtype a;
{
    marquage_mcl=mcl;
    if (last≠0 && mcl==chase)
    {
```

```
        Ens_ote(last,candidats);
        envoi(last,mcl,dfs,p,a,init_mesg);
    }
    else if ((last=Ens_alea(candidats))==0)
    {
            if (pere≠0)
            {
                last=pere;
                envoi(last,mcl,dfs_back,p,a,init_mesg);
            }
            else
                termine_porteur=VRAI;
        }
        else
        {
            Ens_ote(last,candidats);
            envoi(last,mcl,dfs,p,a,init_mesg);
        }
}
FIN

Octet termine_porteur()
{
    return(termine_porteur);
}
FIN

/*————————————————————————————*/
Octet traite_dfs()
{
    if (phase_mesg>phase)
    {
        marquage_dom=init_mesg;marquage_mcl=Mcl;
        if (Mcl==annex)
            Ens_copy(non_cousins,candidats);
        else
            Ens_copy(Voisins,candidats);
        Ens_ote(Org,candidats);pere=Org;
        return(VRAI);
    }

    switch(mocl_mesg)
    {
    case end :
        return(VRAI);
    case dfs :
```

```
        if (marquage_dom≠init_mesg || marquage_mcl ≠ Mcl)
            return(VRAI);
        else if ((Mcl==chase && chased==phase_mesg && candidat<phase)||
                 (Mcl≠chase && chased== -1))
        {
          envoi(Org,Mcl,cousin,phase_mesg,dom_mesg,init_mesg);
          last=Org;
          if (Mcl==annex)
              Ens_ote(Org,non_cousins);
          Ens_ote(Org,candidats);
          return(FAUX);
        }
        else
        {
           if (Mcl==annex)
             Ens_copy(non_cousins,candidats);
           else
             Ens_copy(Voisins,candidats);
           Ens_ote(Org,candidats);
           return(VRAI);
        }
    case cousin:
        if (marquage_dom ≠ init_mesg || marquage_mcl ≠ Mcl)
        {
            /* Un autre message est arrive entre temps qui a change le statut*/
            return(VRAI);
        }
        if (Mcl== annex)
            Ens_ote(Org,non_cousins);
        Ens_ote(Org,candidats);
             /* Pas de break : On continue */
    case dfs_back:
        if (marquage_dom ≠ init_mesg || marquage_mcl ≠ Mcl)
            return(VRAI);                /* cas de noeuds avec un seul voisin. */
        if (Mcl==chase)
            return(VRAI);
        if ((last=Ens_alea(candidats))==0)
            if (pere==0)
            {
                termine_porteur=VRAI;        /* i.e. bientot tout sera termine */
                return(VRAI);
            }
            else
                envoi(last=pere,Mcl,dfs_back,phase_mesg,dom_mesg,init_mesg);
        else
```

```
        {
            Ens_ote(last,candidats);
            envoi(last,Mcl,dfs,phase_mesg,dom_mesg,init_mesg);
        }
        return(FAUX);
    }
}
FIN
/*——————————————————————————*/
Octet defile()
{
  int mocl;
  int p;
  unsigned d,i;
    if (Defile())
    {
        sscanf(Mesg,FORMAT_MSG,&mocl,&p,&d,&i);
        mocl_mesg= mocl; phase_mesg= p;
        dom_mesg= (Pidtype)d;
        init_mesg= (Pidtype)i;
        if (phase_mesg<phase)                          /*retardataire !*/
            return(FAUX);
        else
            return(traite_dfs());
    }
    else
        return (FAUX);
}
FIN
/*——————————————————————————*/
/* FIN des fonctions specif. porteur, debut des fonctions generales */
void nouv_phase()
{
    phase++;dom=Mypid;candidat= -1;chased= -1;
    init(annex,phase,dom);
}
FIN
void maj()
{
    phase= phase_mesg;dom =dom_mesg;candidat= -1;chased= -1;
    cont(annex,phase,dom);
}
FIN
```

```
/*————————————————————————————*/
CALCUL                                                    /* LE programme */
int i;

  1:
     phase= -1;
     Ens_copy(Voisins,non_cousins);
     if (1)                    /* (random() & 01 ) pour des inits quelconques */
     {
        nouv_phase();sleep=FAUX;Snapshot=VRAI;
     }
     else
        sleep=VRAI;
     Va_en 2;
     break;

  2:
     if (defile())
          BRANCH_ON Mcl;
     break;

  ON_MESG end :
    Ens_copy(non_cousins,candidats);
    Ens_ote(Org,candidats);
    Ens_appli(env_end,candidats);
/* transformation du dfs en bfs ! pour faire un AC*/
    THE END;

  ON_MESG annex:                  /* reception d'un jeton en mode annexing */
     if(sleep)
     {
          sleep=FAUX;
          maj();
     }
     else if (termine_porteur())                          /* i.e. terminaison */
     {
          printf("Elu:%d,phase=%d\n",Mypid,phase);
          Ens_appli(env_end,non_cousins);
          THE END;
     }
     else if (phase_mesg>phase ||
               (phase_mesg==phase && dom_mesg==dom && chased<phase))
        maj();
     else if (candidat == phase)
        nouv_phase();
     else if (phase_mesg == phase && (chased == phase || dom>dom_mesg))
```

```
        candidat=phase;
    else if (phase_mesg == phase && dom < dom_mesg && chased < phase)
    {
        chased=phase;
        init(chase,phase,dom);
    }
    Va_en 2;
    break;
  ON_MESG chase:                /* reception d'un jeton en mode chasing */
    if (phase_mesg==phase && dom==dom_mesg && chased<phase &&
candidat<phase)
    {
         chased=phase;cont(chase,phase,dom);
    }
    else if (candidat == phase)
        nouv_phase();
    else if (phase_mesg≥ phase)
    {
        phase =phase_mesg;
        candidat=phase_mesg;Snapshot=VRAI;
    }
    Va_en 2;
    break;
FIN
```

Bibliographie

[AnLe81] Anderson (T.) et Lee (P. A.). – *Fault Tolerance, Principles and Practice.* – Prentice Hall International, 1981.

[AwBG90] Awerbuch (Baruch), Bar-Noy (Amotz) et Gopal (P. M.). – *Approximate distributed Bellman-Ford algorithms.* – RR n° RC 16257, IBM Research Division, 1990.

[Awer85b] Awerbuch (Baruch). – « Reducing complexities of the distributed max-flow and breadth-first-search algorithms by means of network synchronization ». *Networks*, vol. 15, 1985, pp. 425–437.

[Awer85c] Awerbuch (Baruch). – « Complexity of network synchronization ». *Journal of the ACM*, vol. 32, n° 4, 1985, pp. 804–823.

[Awer87] Awerbuch (Baruch). – « Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems ». *ACM Symposium on Theory Of Computing*, pp. 230–240. – 1987.

[AwGa85] Awerbuch (Baruch) et Gallager (Robert G.). – « Distributed BFS algorithms ». *IEEE Symposium on Foundations Of Computer Science*, pp. 250–256. – 1985. (Extended abstract).

[BeDQ86] Bermond (Jean-Claude), Delorme (Charles) et Quisquater (J.-J.). – « Strategies for interconnection networks: Some methods from graph theory ». *Journal of Parallel and Distributed Computing*, vol. 3, 1986, pp. 433–449.

[Berg70] Berge (Claude). – *Graphes et Hypergraphes.* – Paris, Dunod, 1970, *Monographies universitaires de mathématiques*. English translation : Graphs and Hypergraphs (North-Holland, Amsterdam 1973).

[Birk67] Birkoff (G.). – « Lattice theory ». *American Mathematical Society*, vol. 25, 1967.

[BoKo90] Bouabdallah (Abdelmadjid) et König (Jean-Claude). – *Embedding de Bruijn networks in the hypercube.* – RR n° 606, PARIS 11, Orsay, novembre 1990.

[Boll84] Bollobás (Béla). – « The evolution of sparse graphs ». *Graph theory and Combinatorics.* pp. 35–57. – Academic Press, 1984. Proc. Cambridge Combinatorial Conf. in honour of Paul Erdös.

[Boug87] Bougé (Luc). – *Modularité et Symétrie pour les Systèmes Répartis; Application au Langage CSP.* – RR n° 87-2, Labo. d'Informatique de l'Ecole Normale Supérieure, mars 1987. Thèse d'état de l'université Paris 7.

[Brui46] de Bruijn (N. G.). – « A combinatorial problem ». *Koninkl. Nederl. Acad. Wetensch. Proc. Ser*, vol. 49, 1946, pp. 758–764.

[BuBu93] Bui (Marc) et Butelle (Franck). – *Construction répartie d'arbre couvrant de diamètre minimum.* – RR n° 2017, Institut National de Recherche en Informatique et en Automatique, septembre 1993. Version anglaise proposée à la revue Information Processing Letters.

[BuBu93b] Bui (Marc) et Butelle (Franck). – « Minimum diameter spanning tree ». *OPOPAC, Int. Workshop on Principles of Parallel Computing.* pp. 37–46. – Hermès & Inria, novembre 1993.

[Bui89] Bui (Marc). – *Etude Comportementale d'algorithmes distribués de contrôle.* – Thèse de doctorat, Université Paris XI, Orsay, 1989.

[Bui90a] Bui (Marc). – « On optimal management of resources in distributed networks ». *Optimization*, vol. 21, n° 5, 1990.

[Bui94] Bui (Alain). – *Évaluation d'Algorithmes Distribués.* – Thèse de doctorat, Université Paris 7, 1994. En cours.

[Burn81] Burns (J. E.). – « Symmetry in systems of asynchronous processes ». *22nd symp. on Foundation of Computer Science*, pp. 169–174. – 1981.

[Bute93] Butelle (Franck). – « Election répartie non-prédéterminée ». *Réseaux et Informatique Répartie*, vol. 3, n° 2, 1993, pp. 125–143.

[CaGM80] Camerini (P. M.), Galbiati (G.) et Maffioli (F.). – « Complexity of spanning tree problems: Part I ». *European Journal of Operationnal Research*, vol. 5, 1980, pp. 346–352.

[CAPA93] CAPA93. – Ecole d'automne : Conception et analyse d'algorithmes parallèles. – à paraitre chez Hermès, 1993.

[CASD85] Cristian (Flaviu), Aghili (H.), Strong (Ray) et Dolev (Danny). – « Atomic broadcast: from simple message diffusion to Byzantine agreement ». *FTCS, Int. symp. on Fault Tolerant Computing.* – 1985.

[Chri75] Christophides (Nicos). – *Graph Theory: An algorithmic approach.* – Academic press, 1975, *Computer Science and Applied Mathematics.*

[ChRo79] Chang (Edwards J.) et Roberts (Robert). – « An improved algorithm for decentralized extremum–finding in circular configurations ». *Communications of the ACM*, vol. 22, n° 5, 1979, pp. 281–283.

[Dijk59] Dijkstra (Edsger W.). – « A note on two problems in connection with graphs ». *Nümerishe Mathematik*, vol. 1, 1959, pp. 269–271.

[Duni92] Dunigan (T.). – *Kendall Square Multiprocessor: Early Experiences and Performance.* – ORNL/TM n° 12065, Oak Ridge National Laboratory, 1992.

[FiLP85] Fisher (Michael J.), Lynch (Nancy A.) et Paterson (Michael S.). – « Impossibility of distributed consensus with one faulty process ». *Journal of the ACM*, vol. 32, n° 2, 1985, pp. 374–382.

[FlGL93] Florin (Gérard), Gómez (Roberto) et Lavallée (Ivan). – « Recursive distributed programming schemes ». *Int. Symp. on Autonomous Decentralized Syst.* pp. 122–128. – IEEE Computer Society Press, 1993.

[Flyn66] Flynn (Michael J.). – « Very high–speed computing systems ». *Proc. of the IEEE*, vol. 54, n° 12, 1966, pp. 1901–1909.

[FMRT90] Fisher (M. J.), Moran (Shlomo), Rudich (S.) et Taubenfeld (G.). – « The wakeup problem ». *ACM Symposium on Theory Of Computing*, pp. 106–116. – 1990. (Extended abstract).

[Fred85] Frederickson (Greg N.). – « A single source shortest path algorithm for a planar distributed network ». *Symposium on Theoretical Aspects of Computer Science.* pp. 143–150. – Springer Verlag, 1985.

[FrGS90] Froidevaux (Christine), Gaudel (Marie-Claude) et Soria (Michèle). – *Types de données et algorithmes.* – McGraw-Hill, 1990.

[GaHS83] Gallager (Robert G.), Humblet (Pierre A.) et Spira (Paul M.). – « A distributed algorithm for minimum weight spanning trees ». *ACM Transactions on Programming Languages and Systems*, vol. 5, n° 1, 1983, pp. 66–77.

[GaJo79] Garey (Michael R.) et Johnson (David S.). – *Computers and intractability: A guide to the theory of NP-completeness.* – W. H. Freeman, 1979.

[Gayr92] Gayraud (Thierry). – *Contribution à la résolution de problèmes d'optimisation dans les graphes par des algorithmes parallèles.* – Thèse de doctorat, Université Paul Sabatier de Toulouse, oct 1992.

[GoHK91] Goldfarb (D.), Hao (J.) et Kai (S.-R.). – « Shortest path algorithms using dynamic breadth-first search ». *Networks*, vol. 21, n° 1, 1991, pp. 29–50.

[GoLe91] Gornet (Jean-François) et Levilion (Marc). – « Etat de la normalisation OSI ». *Réseaux et Informatique Répartie*, vol. 1, n° 3, 1991, pp. 297–309.

[GrHe85] Graham (Ronald L.) et Hell (Pavol). – « On the history of the minimum spanning tree problem ». *Annals of the History of Computing*, vol. 7, n° 1, 1985, pp. 43–57.

[HeMR87] Hélary (Jean-Michel), Maddi (Aomar) et Raynal (Michel). – « Calcul réparti d'un extrémum et du routage associé dans un réseau quelconque ». *RAIRO Informatique théorique et applications*, vol. 21, n° 3, 1987, pp. 223–.

[HeRa87] Hélary (Jean-Michel) et Raynal (Michel). – *Deapth-First Traversal and Virtual Ring Construction in Distributed Systems.* – RR n° 704, Institut National de Recherche en Informatique et en Automatique, 1987.

[HLCW91] Ho (J.-M.), Lee (D. T.), Chang (C.-H.) et Wong (C. K.). – « Minimum diameter spanning trees and related problems ». *SIAM Journal on Computing*, vol. 20, n° 5, octobre 1991, pp. 987–997.

[Hoar78] Hoare (Charles A. R.). – « Communicating sequential processes ». *Communications of the ACM*, vol. 21, n° 8, 1978, pp. 666–677.

[Hoar87] Hoare (Charles A. R.). – *Processus séquentiels communicants.* – Masson, 1987.

[Humb91] Humblet (Pierre A.). – « Another adaptative distributed shortest path algorithm ». *IEEE Transactions on Computers*, vol. 39, n° 6, 1991, pp. 995–1003.

[IhRW91] Ihler (E.), Reich (G.) et Wildmayer (P.). – « On shortest networks for classes of points in the plane ». *Int. Workshop on Computational Geometry – Meth., Alg. and Appli.*, pp. 103–111. – mars 1991.

[INMOS87] INMOS, SGSS-Thomson Microelectronics. – *IMS T800 transputer: engeneering data*, avril 1987.

[KoKM90] Korach (Ephraïm), Kutten (Shay) et Moran (Shlomo). – « A modular technique for the design of efficient distributed leader finding algorithms ». *ACM Transactions on Programming Languages and Systems*, vol. 12, n° 1, 1990, pp. 84–101.

[Krus56] Kruskal (J. B.). – « On the shortest spanning subtree of a graph and the travelling salesman problem ». *Proc. of the AMS*, pp. 48–50. – 1956.

[LADF+93] Leiserson (C. E.), Abuhamdeh (Z. S.), Douglas (D. C.), Feynman (C. R.), Ganmukhi (M. N.), Hill (J. V.), Hillis (W. D.) et al. – *The Network Architecture of the Connection Machine CM-5*. – Rapport technique n° 02142, Thinking Machines Corporation – Cambridge, Massachussetts, août 1993.

[LaLa86] Lavallée (Ivan) et Lavault (Christian). – « Scheme for efficiency performance measures of distributed and parallel algorithms ». *Proc. of the 1st Int. Workshop on Distributed Algorithms, Distributed algorithms on graphs*, éd. par Gafni (E.) et Santoro (N.). pp. 69–103. – Carleton University Press, 1986.

[LaLa89a] Lavallée (Ivan) et Lavault (Christian). – *Yet another distributed election and (minimum-weight) spanning tree algorithm*. – RR n° 1024, Institut National de Recherche en Informatique et en Automatique, 1989.

[LaLa91] Lavallée (Ivan) et Lavault (Christian). – « Spanning tree construction for nameless networks ». *4thWorkshop on Distributed AlGorithm*. pp. 41–56. – Springer Verlag, 1991.

[Lamp78] Lamport (Leslie). – « Time, clocks and the ordering of events in a distributed system ». *Communications of the ACM*, vol. 21, n° 7, 1978, pp. 558–565.

[Lapr92] Laprie (Jean-Claude). – « Dependability: basic concepts and terminology ». *FTCS, Int. symp. on Fault Tolerant Computing*, éd. par Springer-Verlag. – 1992.

[LaRo86] Lavallée (Ivan) et Roucairol (Gérard). – « A fully distributed (minimal) spanning tree algorithm ». *Information Processing Letters*, vol. 23, 1986, pp. 55–62.

[LaSP82] Lamport (Leslie), Shostak (R.) et Pease (M.). – « The byzantine generals problem ». *ACM Transactions on Programming Languages and Systems*, vol. 4, n° 3, juillet 1982, pp. 382–401.

[Laval86] Lavallée (Ivan). – *Contribution à l'algorithmique parallèle et distribuée, application à l'optimisation combinatoire.* – Thèse d'état, Université Paris XI, Orsay, 1986.

[Laval90] Lavallée (Ivan). – *Algorithmique parallèle et distribuée.* – Hermès, 1990.

[Lavau87] Lavault (Christian). – *Algorithmique et complexite distribuées - application.* – Thèse d'état, Univeristé Paris-Sud Orsay, 1987.

[Lavau90a] Lavault (Christian). – *Theoretical Computer Science73*, chap. Exact average message complexity values for distributed election on bidirectional rings of processors, pp. 61–79. – North-Holland, 1990.

[LCGW+93] James E. Lumpp (Jr), Casavant (Thomas L.), Gannon (Julie A.), Williams (Kyle J.) et Andesland (Mark S.). – « Analysis of communication patterns for modeling message passing systems ». *OPOPAC, Int. Workshop on Principles of Parallel Computing.* pp. 249–258. – Hermès & Inria, novembre 1993.

[Leis85] Leiserson (C. E.). – « Fat-Trees: Universal networks for hardware-efficient supercomputing ». *IEEE Transactions on Computers*, vol. 34, 1985, pp. 892–901.

[Lela77] Le Lann (Gérard). – « Distributed systems : towards a formal approach ». *Proc. of IFIP congress 77.* IFIP Congress, pp. 155–160. – DEC – Systems Research Center, Palo Alto Californie 1977.

[Lync89] Lynch (Nancy). – *A hundred impossibility proofs for distributed computing.* – MIT/LCS/TM n° 394, Massachusetts Institute of Technology, août 1989.

[Marg89] Margulis (S.). – « The intel 80860 ». *Byte*, 1989.

[Matt87] Mattern (Friedemann). – « Algorithms for distributed termination detection ». *Distributed Computing*, vol. 2, n° 3, 1987, pp. 161–176.

[MeSe79] Merlin (Philip M.) et Segall (Adrian). – « A failsafe distributed routing protocol ». *IEEE Transactions on Computers*, vol. COM-27, n° 9, septembre 1979, pp. 1280–1287.

[NaTA88] Naïmi (Mohammed), Tréhel (Michel) et Arnold (André). – *A* log n *Distributed Mutual Exclusion Algorithm Based on the Path Reversal.* – RR, Laboratoire d'Informatique de Besançon (France), avril 1988.

[Palm92] Palmer (J.) et G. L. S. (jr). – « Connection machine model CM-5 overview ». *4th Symposium on the Frontiers of Massively Parallel Computation.* pp. 474–483. – IEEE Computer Society Press, 1992.

[Pele90] Peleg (David). – « Time-optimal leader election in general networks ». *Journal of Parallel and Distributed Computing*, vol. 8, n° 1, 1990, pp. 96–99.

[Prim57] Prim (R. C.). – « Shortest connections networks and some generalizations ». *Bell System Technical Journal*, vol. 36, 1957, pp. 1389–1401.

[RaHe88] Raynal (Michel) et Helary (Jean-Michel). – *Synchronisation et contrôle des systèmes et des programmes répartis.* – Eyrolles, 1988.

[Rayn87] Raynal (Michel). – *Systèmes répartis et réseaux.* – Eyrolles, 1987.

[Rayn91] Raynal (Michel). – *La communication et le temps dans les réseaux et les sytèmes répartis.* – Eyrolles, 1991.

[Roli89] Rolin (Pierre). – *Réseaux locaux : normes et protocoles.* – Hermès, 1989, *Traité des nouvelles technologies Série automatique.* 2e ed. rev. et augm.

[Rume93] de Rumeur (Jean). – *école d'été 1992 sur les communications dans les réseaux de processeurs.* – Masson, 1993.

[Sanc93] Sanchez (Jesús). – *Prédiction et évaluation de performace des algorithmes adaptatifs implantés sur machines parallèles.* – Thèse de doctorat, École Nationale Supérieure des Télécommunications, 1993.

[Sans88] Sansonnet (Jean-Paul). – « L'architecture des nouveaux ordinateurs ». *La Recherche*, novembre 1988, pp. 1298–1307.

[Sans89] Sansonnet (Jean-Paul). – Concepts d'architectures avancées. partie 2: Architectures massivement parallèles et multiprocesseurs. – LRI, 1989.

[Sega83] Segall (Adrian). – « Distributed network protocols ». *IEEE Transactions on Information Theory*, vol. IT-29, n° 1, janvier 1983, pp. 23–35.

[Soll61] Sollin (Georges). – Exposé du séminaire de C. Berge, I.H.P., 1961. Repris in extenso dans *Méthodes et modèles de la Recherche Opérationnelle*,T.2, Kauffmann (ed. Dunod, Paris 1963), pp. 33–45.

[Tane90] Tanenbaum (Andrew). – *Réseaux : architecture, protocoles et applications.* – InterEditions, 1990.

[Tarj83] Tarjan (Robert E.). – *Data structures and network algorithms.* – SIAM, 1983.

[Tel91] Tel (G.). – *Topics in distributed computing.* – Thèse de PhD, Utrecht, 1991.

[TEXAS92] Texas Instruments. – *TMS320C4x. User's guide*, 1992.

[TrNa87] Tréhel (Michel) et Naïmi (Mohammed). – « Un algorithme distribué d'exclusion mutuelle en $\log(n)$ ». *Technique et Science Informatiques*, vol. 6, n° 2, 1987, pp. 141–150.

[Yao75] Yao (Andrew Chi-Chih). – « An $O(|E| \log\log |V|)$ algorithm for finding minimum spanning trees ». *Information Processing Letters*, vol. 4, 1975, pp. 21–25.

Table des figures

Table des matières