

Un algorithme probabiliste d'election et d'arbre couvrant sur des reseaux anonymes

Ivan Lavallee, C. Lavault

▶ **To cite this version:**

Ivan Lavallee, C. Lavault. Un algorithme probabiliste d'election et d'arbre couvrant sur des reseaux anonymes. RR-1151, INRIA. 1989. inria-00075408

HAL Id: inria-00075408
https://inria.hal.science/inria-00075408

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1151

Programme 2
Structures Nouvelles d'Ordinateurs

UN ALGORITHME PROBABILISTE D'ÉLECTION ET D'ARBRE COUVRANT SUR DES RÉSEAUX ANONYMES

Ivan LAVALLÉE
Christian LAVAULT

Décembre 1989

RR_1151

A PROBABILISTIC ELECTION AND SPANNING TREE ALGORITHM IN ANONYMOUS NETWORKS

UN ALGORITHME PROBABILISTE D'ÉLECTION ET D'ARBRE COUVRANT SUR DES RÉSEAUX ANONYMES

Ivan LAVALLÉE[1] *Christian* LAVAULT[2]

Mots clés: Élection, arbre couvrant, algorithmes probabilistes, terminaison distribuée, asynchronisme.

Key words: Leader election, spanning tree, probabilistic algorithms, termination detection, asynchrony.

[1]*INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex. France.*
e-mail : lavallee@seti.inria.fr
Département d'Informatique, Université Paris VIII, France.

[2]*INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex. France.*
e-mail : lavault@seti.inria.fr
Département d'Informatique, Université Paris XII, France.

Ce travail a été réalisé dans le cadre du projet *CAPRAN* de l'INRIA, avec le soutien de C-cul *COPARADIS*).

PAPIER RÉCUPÉRÉ ET RECYCLÉ

Abstract

Two types of distributed asynchronous probabilistic algorithms are given in the present report which elect a leader and find a spanning tree in arbitrary anonymous networks of processes. So far, these algorithms are very likely the first distributed election algorithms for nameless networks to be fully and occurately specified.

They are basically patterned upon a previous deterministic spanning tree algorithm for general networks whith distinct processes' identities designed in [LA,LA-89]. Two different cases are considered here. First, we adress the case when strictly no global information on the network is a priori assumed for processes. This leads to such a type of probabilistic algorithm where only a fixed given number of random draws is involved; there is no termination detection for this case and a unique leader and spanning tree be found with probability $\geq 1 - \varepsilon$, for some given $\varepsilon > 0$. Second, knowledge of the size n of the network is assumed for a priori at least one process. Then the number of random draws must not be fixed in advance; the election and the spanning tree problems are both solved with termination detection and without error, since the termination of the algorithm is completed whenever one process learns that it belongs to same tree of size n which thus spans the whole network.

Résumé

Dans le présent rapport, nous proposons deux variantes d'un algorithme distribué, probabiliste, asynchrone d'élection et de construction d'arbre couvrant dans des réseaux anonymes à topologie quelconque. A notre connaissance, cet algorithme est le premier du genre, à être totalement et rigoureusement spécifié.

Il est fondé sur un précédent algorithme déterministe d'élection pour réseaux de processus à identités distinctes (voir [LA,LA-89]). Nous montrons ici comment construire les deux variantes, suivant deux contextes différents. Premièrement, nous considérons le cas ou on ne possède strictement aucune connaissance globale sur le réseau, ce qui conduit à un algorithme dans lequel on n'opère qu'un nombre limité, fixé d'avance, de tirages aléatoires. Il n'y a pas alors de moyen déterministe d'obtenir une "bonne" terminaison distribuée, et l'élection comme la construction d'un arbre couvrant sont réalisés avec une probabilité $\geq 1 - \varepsilon$, pour $\varepsilon > 0$ donné. Et deuxièmement, en supposant qu'un processus au moins connaisse n le nombre total de processus du système, le problème de l'élection et de l'arbre couvrant sont résolus avec une terminaison correcte de l'algorithme, sans erreur. La terminaison correcte de l'algorithme tient ici au fait que l'un des processus finit par apprendre qu'il appartient à un arbre à n sommets; cet arbre est alors bien un arbre couvrant du réseau.

1. Introduction

Le problème de l'élection est un des thèmes rémanents de l'algorithmique distribuée depuis de nombreuses années [LEL-77]. Du point de vue du contrôle des systèmes répartis, l'**élection**, et la construction d'un **arbre couvrant**, sont des problèmes fondamentaux. En effet, l'élection fournit le moyen d'attribuer un privilège (désignation d'un coordinateur, régénération de jeton, etc.) à un processus, et l'arbre couvrant permet de structurer l'ensemble des processus considérés en fournissant un système de contrôle et de routage des messages au sein du réseau de communication. Ces deux problèmes, élection et construction d'un arbre couvrant, sont à la base des principaux mécanismes de contrôle utilisés dans les systèmes répartis, tels l'exclusion mutuelle, la synchronisation, le redémarrage sur panne, le routage, etc.

Tous les algorithmes opérant par comparaison entre valeurs qui traitent de ce problème utilisent fondamentalement le fait que les processus ont des *identités distinctes*. Cette classe d'algorithmes d'élection est essentiellement composée d'algorithmes de recherche d'un extremum, de détection du processus d'identité maximale ou minimale. Sur le plan théorique, le problème de l'élection dans un réseau synchrone ou asynchrone est d'ailleurs, par essence, *sans solution exacte lorsque les identités des processus ne sont pas toutes distinctes* ; il est alors impossible de "briser la symétrie" structurelle des processus du système [ANG-80].

Ici au contraire, nous considérerons qu'au départ, tout ou partie des processus possède une même identité (ou, ce qui revient au même, que ces processus n'ont pas d'identité du tout). C'est la situation qu'on peut trouver après une panne générale ou partielle du réseau. L'élection consiste alors à passer d'un système réparti caractérisé par le fait que tous les processus sont dans un même état (*candidat*) et qu'un certain nombre d'entre eux, voire tous, ont des identités non distinctes, à un système dans lequel un seul et unique processus est dans l'état *élu*, et à construire le routage correspondant.

Comme dans le cas d'une solution déterministe [LA,LA-89], la construction d'un arbre couvrant du graphe sous-jacent au réseau se fait progressivement par le maintien d'une forêt couvrante, grâce à la conservation d'une propriété d'absence de cycle entre sous-arbres (*fragments*). Ainsi, La taille de ces fragments va croissant et, partant, leur nombre diminue au cours du temps, jusqu'à la construction d'un arbre couvrant unique solution du problème. La racine de ce dernier est alors le site ou sommet-processus élu et le problème de l'élection trouve sa solution dans celle de la construction de l'arbre couvrant.

Mais, dans le cas présent, comme on l'a souligné au paragraphe précédent, ces problèmes ne peuvent être résolus par un algorithme simplement déterministe comme dans [GA,HU,SP-85],

[LA,RO-86],[AWE-88] ou [LA,LA-89]. Lorsque les identités des processus ne sont pas toutes distinctes, la symétrie est impossible à briser et le problème est sans solution. Seuls des algorithmes probabilistes permettent de contourner cette difficulté en donnant une solution partielle très acceptable au sens probabiliste à ces deux problèmes.

2. Le problème.

2.1 Hypothèses

Un système réparti, ou réseau est un ensemble de n sites ou processeurs reliés entre eux, deux à deux par des lignes de communications. Nous considérerons l'abstraction logique d'un tel système comme étant un algorithme distribué. Un algorithme distribué est alors un ensemble de processus séquentiels communicant entre eux par l'intermédiaire de messages. On peut associer à tout algorithme distribué un graphe dont les sommets sont les processus de l'algorithme, et dont les arêtes figurent les possibilités de communications directes entre les processus, c'est à dire les lignes de communication point à point du système réparti. Par abus de langage, on utilisera les termes sommet du graphe, ou processus dans le même sens dans la mesure où aucune ambiguïté n'est possible; de même pour arête et ligne de communication. Chaque processus possède une mémoire locale non partageable, une horloge locale (ou pas d'horloge du tout si on veut), et ne peut communiquer avec ses voisins que par émission et réception de messages. La façon dont s'effectue la transmission des messages amène à distinguer les réseaux en *synchrones* et *asynchrones*. Dans un système asynchrone, les délais de transit des messages sur les lignes de communication sont finis, mais non bornés, imprévisibles. Dans les réseaux synchrones, toutes les horloges sont supposées battre de manière synchrone, et tout message émis au temps t est reçu au temps $t+1$.

2.2 Résultats

De nombreux algorithmes d'arbre couvrant ont été proposés sous l'hypothèse d'identités des processus toutes distinctes, [GA,HU,SP-85],[LAV-88], [AWE-88] et [LA,LA-89] en particulier proposent des algorithmes de complexité en messages $\Theta(e + n\log n)$ optimale dans le pire des cas, l'algorithme d'Awerbuch étant aussi optimal en temps, à savoir $\Theta(n)$. La plupart des algorithmes de ce type utilisent la propriété de Sollin [SOL-61] (cf. Boruvka, Choquet et Lukaszewicz et al., puis Kruskal et enfin Jarnik, Prim et Dijkstra etc.) et une relation d'ordre stricte entre les fragments (basée sur une notion de niveau et des mécanismes empêchant la fusion simultanée de plus de deux fragments). A l'exception de [LAV-88] et [LA,LA-89], ces algorithmes utilisent tous la notion fondamentale de niveau afin d'obtenir une croissance équilibrée des fragments construits, deux fragments ne fusionnant que s'ils sont au "même niveau" (i.e. si leurs tailles sont égales). C'est là une méthode de pseudo-synchronisation des algorithmes, et une première façon d'obtenir l'optimalité en nombre de messages.

Ce mécanisme de fusion à niveau égal est remplacé dans [LA,LA-89] par un autre mécanisme, absolument différent, qui gère de manière asynchrone le passage des messages dans les arêtes sortantes des fragments. Cette seconde méthode permet d'assurer des successions dynamiques de fusions asynchrones entre fragments qui s'opèrent de manière optimale, sans pour autant pseudo-synchroniser le processus et le contraindre à une croissance équilibrée, par niveaux.

L'algorithme ici présenté est inspiré des algorithmes de construction déterministe d'arbre couvrant proposés dans [LA,RO-86], [LAV-88] et [LA,LA-89]. Toutefois, cet algorithme ne reprend que partiellement le mécanisme de fusion ci-dessus évoqué pour tenir compte de l'aspect aléatoire de certaines informations. On pourrait réintroduire ce mécanisme dans sa totalité, mais cela compliquerait notablement la spécification de l'algorithme sans rien apporter d'essentiel sur la compréhension de celui-ci, aussi avons nous préféré sacrifier à la clarté de l'exposé.

Cependant, cet algorithme est probabiliste et utilise un procédé de gestion des égalités entre identités par un tirage au sort initial (absence d'identités) de celles-ci sur un ensemble totalement ordonné I de grande taille, $d = O(r.\lg r)$ (voir plus loin la valeur de r). Ensuite, les égalités possibles entre identités avec une probabilité très faible (car $d = O(r.\lg r)$), sont brisées par un retirage aléatoire sur le même ensemble de taille d selon une fonction monotone de l'identité

L'algorithme peut ainsi fonctionner selon la méthode dite "de Monte-Carlo", c'est à dire se terminer avec une solution "incorrecte", - une forêt couvrante - ou bien avec la solution correcte, à savoir un arbre couvrant. Pour toute instance du problème, la solution correcte s'obtient avec une probabilité supérieure à $1-\varepsilon$ alors que la solution "incorrecte" n'a qu'une très faible probabilité d'apparition.

Il peut aussi fonctionner selon la méthode dite "de Las Végas" dont la caractéristique est de prendre arbitrairement des décisions aléatoirement "sans solution correcte", c'est à dire de renvoyer comme solution soit la construction correcte d'un arbre couvrant, soit une solution incorrecte - en l'occurrence une forêt couvrante. Mais dans ce second cas, il suffit de soumettre à nouveau cette même instance à ce même algorithme pour avoir une autre possibilité d'arriver à la solution. Au total, d'exécution en exécution, la probabilité de succès de construction d'un arbre couvrant augmente vite, jusqu'à une valeur très voisine de 1. Pour cette seconde catégorie d'algorithmes, on ne peut à proprement parler utiliser le concept de terminaison ; un certain nombre r de réexécutions étant fixé arbitrairement au départ, l'algorithme s'arrête en donnant une solution au problème de l'arbre couvrant avec une probabilité d'autant plus proche de 1 que le paramètre de réglage r (ou crédit, voir plus loin) est plus grand.

Par ailleurs, pour ces deux types d'algorithmes, la connaissance globale de n conférée à tous les processus n'est pas nécessaire à la solution du problème. Elle permet d'obtenir une "bonne" terminaison simplifiée et plus facile à démontrer. Elle permet de plus de calculer d en fonction de n

dès le début de l'algorithme ; dans ce cas, on prend d comme majorant de n ; autrement dit, on effectue les tirages aléatoires sur un ensemble $\boldsymbol{I}$ de cardinal d beaucoup plus grand que n. Ainsi, en prenant $d \geq n^2/\varepsilon$, la probabilité que deux identités quelconques tirées aléatoirement sur $\boldsymbol{I}$ soient différentes est $\geq 1 - \varepsilon$, pour tout $\varepsilon > 0$ fixé.

Les récentes solutions probabilistes au problème de l'élection et de l'arbre couvrant proposées par Afek & Saks [AF,SA-87] et par Matias & Afek [MA,AF-89] ouvrent la voie à la conception d'algorithmes probabilistes trés performants. Ainsi, dans [MA,AF-89], l'algorithme distribué d'élection anonyme fonctionne avec une probabilité d'erreur ε en utilisant $O(m\lg n \cdot r\lg r)$ messages ($r = 1/\varepsilon$) dans le pire des cas et sans connaissance de n, les messages étant de taille $O(\lg\lg n + \lg r)$ et la complexité en temps $O(D)$ où D est le diamètre du réseau. Lorsque n est connu, $N < n < kN$ ($k \geq 1$), les résultats sont bien meilleurs, à savoir : $O(mr\lg(rk)\lg r)$, $O(\lg n)$ et $O(D)$, respectivement.

L' algorithme que nous proposons ici, et sa variante, a des performances comparables pour la complexité en messages et en temps. Mais surtout, il s'agit à notre connaissance du premier algorithme probabiliste d'élection et d'arbre couvrant anonyme qui soit rigoureusement spécifié, les solutions citées plus haut étant présentées comme des schémas algorithmiques plutôt que comme de véritables algorithmes.

3. L'algorithme.

Dans ce qui suit nous adopterons le modèle de système réparti décrit ci-dessus. Nous supposerons de plus que ce système est muni d'une couche basse de logiciel qui lui assure un système postal ayant les propriétés suivantes:

- Tout message envoyé du sommet-processus a sur l'arète-ligne (a,b) arrive au bout un temps fini, mais non prévisible au sommet processus b.

- Les messages ne se dépassent pas sur une même ligne, l'ordre d'arrivée des messages sur une ligne est le même que leur ordre d'émission.

- Il n'y a ni perte ni altération, ni duplication de message.

- Outre l'absence d'erreur de transmission de message, on supposera que les files d'attentes d'arrivée des messages sont gérées suivant la discipline PAPS (Premier Arrivé, Premier Servi).

Les processus sont totalement symétriques, au sens où chacun d'eux possède le même code au démarrage, y compris la même identité (ou l'absence d'identité). Sans perte de généralité, on suppose de plus qu'au début de l'algorithme, tous les processus démarrent en même temps.

3.1 Description de l'algorithme.

L'algorithme utilise la propriété locale de Sollin [SOL-61]. Cette propriété permet la conception des algorithmes d'arbre couvrant distribué par croissance simultanée de fragments (sous-arbres) indépendants, c'est ainsi que procèdent la plupart des algorithmes existants La classe des "bons" algorithmes déterministes opérant par comparaisons se caractérise par l'utilisation d'une "bonne" propriété de fusion permettant la *fusion simultanée* de plusieurs fragments, et si possible du plus grand nombre, c'est ce que font [AWE-88]; [GA,HU,SP-85], en introduisant une notion de niveau. C'est aussi le cas dans [LAV-88] et [LA,LA-89], mais sans utiliser de notion de niveau, en intervenant de façon locale sur le sens de parcours des arètes par les messages de requêtes.

Le parallélisme maximum dans les fusions simultanées de fragment est ici obtenu par le fait qu'on inverse le "sens habituel" de sollicitation des fusion. Comme dans [LA,RO-86] un fragment demande l'autorisation de venir se fusionner dans un autre, la réponse à une telle demande étant laissée au premier processus d'un fragment qui la reçoit, il peut y avoir plusieurs accords simultanés d'un même fragment pour que d'autres viennent s'y fusionner.

3.1.1. Le principe de l'algorithme.

Dans [LAV-86], [LAV-88] et [LA,LA-89], la propriété locale assurant la possibilité de fusion entre sous arbres sans créer de cycle, donc conservant la propriété de sous-arbre, est obtenue par référence à la relation d'ordre strict entre les identités des racines. Ici, une telle relation d'ordre strict n'existe pas. Par conséquent, l'algorithme va consister à créer, tant qu'on le pourra une relation d'ordre chaque fois qu'on se trouvera en présence d'une égalité pour laquelle on ne sait pas dire s'il s'agit, ou non, d'un processus du même fragment. La relation d'ordre utilisée est donnée par les numéros d'identification des fragments (i.e. l'identité de leur racine).

3.1.2 Les étapes de la fusion des fragments.

Considérons deux fragments F et G (d'identificateurs f et g) et une arète (a,b) ayant une extrémité, a par exemple, dans F et l'autre, b, dans G. Supposons de plus que le processus a possède, dans F, le privilège d'émettre une demande de fusion (on verra que, par construction, dans F, ce processus est le seul à posséder ce privilège (cf § 3.1.5). Le processus a envoie vers b, une demande de fusion. La fusion de F dans G va dépendre de la relation d'ordre ci-dessus définie.

Si $f < g$, alors la fusion de F dans G sera acceptée, si $f = g$, la fusion ne saurait être acceptée sans risque de créer un, cycle. Pour éviter cela, le processus qui a émis la demande de fusion envoie vers son père une demande de retirage d'identificateur. Ce retirage, effectué par la racine f du fragment F n'aura lieu que si la racine en question possède encore du "crédit", sinon sa réponse sera

cousin. La fusion de F dans G consiste en la constitution d'un fragment d'identificateur g, et dont l'ensemble des sommets est la réunion des ensembles de sommets de F et G.

Le principe de fonctionnement de l'algorithme est alors le suivant. Un processus x d'un fragment (et un seul dans chaque fragment) d'identité de racine i envoie une demande de fusion à un processus y connexe (voisin). Soit j l'identité de la racine du fragment contenant le processus y.

Si on a $i > j$, alors le processus y peut prendre la décision de rejeter la demande, et cela de façon purement locale (de toute façon, l'invariant devant être la propriété de non-cyclage, même si le rejet est abusif, cette propriété est conservée).

Par contre, si $i = j$, deux cas peuvent se produire: soit les processus font partie du même fragment, soit non. Mais dans ce dernier cas, les deux fragments ont des racines de même identité Dans ce cas d'égalité, y renvoie un message de mot clé **equal** à x. Le processus x répercute ce message par l'intermédiaire de son père, vers la racine i de son fragment. Si x est la racine du fragment ($x = i$), x tire une nouvelle identité i' telle que $i' > i$ si son crédit (voir § 3.1.3.) est positif, envoie un message de mise à jour à tous les processus de son fragment et décrémente de une unité son crédit. Si le crédit de x est nul, le tirage est terminé; x considère alors que le message **equal** vient d'un processus du même fragment et envoie donc à ce processus, un message **cousin** par l'intermédiaire de la porte qui avait transmis le message **equal.** Ce message **cousin** permet au processus qui, ayant émis un message **comb** de demande de fusion, avait reçu pour réponse **equal**, de lever l'ambiguité ainsi créée. Dans le cas de réception d'un message **cousin**, le processus en question, procède à une mise à jour de ses variables *CAND* et *open*, et s'il possède toujours des candidats (*open* $\neq$ faux), il peut envoyer de nouveau sur l'un de ceux-ci, un message **comb**. Si le processus ne possède plus de candidat, il transmet le privilège d'émission de message **comb** à son père.

Si on a $i < j$, la décision d'accepter la fusion peut être prise localement. Par conséquent, y renvoie un message de mot clé **ok**. En effet, même si la valeur de j, (l'identificateur de racine du fragment contenant y) est en cours de modification sans que y en soit informé, la nouvelle valeur j' ne peut être que supérieure à j (par construction), et par transitivité, la relation d'ordre est conservée.

3.1.3. La notion de crédit.

La notion de crédit intervient ici comme un paramètre donné à l'avance, de "réglage" de la précision de l'algorithme. En effet, c'est le nombre de fois que l'on peut tirer une nouvelle identité. En fait, intuitivement, on conçoit bien que plus ce crédit est élevé, plus la probabilité d'obtenir un arbre couvrant à la fin de l'exécution de l'algorithme est proche de 1. Il est nécessaire de fixer cette valeur, car en l'absence de toute information de caractère global sur le système réparti (connaissance

de n -nombre de processus- par exemple), c'est la valeur r de ce paramètre qui peut permettre de résoudre le problème de l'arbre couvrant avec une probabilité supérieure ou égale à $1 - \varepsilon$, avec $r = 1/\varepsilon$, $\varepsilon > 0$, erreur donnée de l'algorithme de résolution.

3.1.4 La gestion de la circulation des messages.

Plusieurs messages **comb** (c'est le message de demande de fusion) peuvent arriver en même temps dans un fragment, en provenance d'un fragment extérieur (un seul par fragment). Par contre il n'y a, à un instant donné, qu'un seul message **comb** dans un fragment, émis par un processus de celui-ci possédant le privilège de le faire. Pour tout processus d'un fragment recevant le message **comb** émis par le processus du même fragment, ce message ne peut lui parvenir que par son père (c'est dans ce cas, en fait une transmission du privilège), tout autre message **comb** venant par une porte autre que celle correspondant à son père, est considérée comme provenant d'un autre fragment.

Comme dans [LA,RO-86], et plus encore dans [LA,LA-89], la gestion des réponses aux messages de demande de fusion (messages **comb**) est complètement laissée aux processus recevant pour la première fois ce message.

Le problème à résoudre vient de ce que certains messages doivent être transmis à la racine du fragment (c'est le cas des messages **equal**), et qu'il est nécessaire que la réponse éventuelle à ce message suive le même chemin en sens inverse, sans connaissance globale de ce chemin. Il faut par conséquent créer une méthode de chaînage qui permette de caractériser le passage de ces messages de façon à pouvoir retransmettre sur la porte appropriée la réponses propre à chaque message. Cette méthode est assez générale pour permettre de gérer plusieurs transits simultanés de messages de même nature, sans qu'il y ait pour autant confusion. Pour ce faire, en chaque processus, on gère une horloge logique h qu'on incrémente chaque fois qu'on envoie un message vers le père, en transmettant l'heure ainsi calculée dans le message. De plus, on crée un tableau $TABLE[]$indexé par les heures logiques locales, et chaque fois qu'on renvoie un message vers le père, on stocke dans $TABLE[h]$ l'adresse de la porte par laquelle était arrivé le message qu'on retransmet, et l'heure logique qui lui avait été attribuée par le processus l'ayant transmis (cette heure fait partie du message). On a ainsi, dans le processus i, une instruction du type $TABLE[h] := (j, e)$ où h est l'heure, locale à i, à laquelle le message a été transmis au père de i, j est l'adresse de la porte par laquelle i a reçu le message correspondant, et e est l'heure logique, locale au processus correspondant à la porte d'adresse j, à laquelle le message a été envoyé à i. Un processus étant capable de distinguer toutes ses portes et les heures locales à un processus étant toutes différentes, tous les messages sont donc parfaitement identifiés. Ainsi lorsque le processus i recevra une réponse à un message, ce ne peut être qu'à un message qu'il avait lui-même transmis; cette réponse ne peut lui parvenir que par son père (sinon, c'est que c'est i lui même qui a créé le message, et la réponse est pour lui), et comme la

réponse contient l'heure logique h à laquelle il avait transmis le message, il retrouve dans *TABLE*[h] l'adresse de la porte sur laquelle il lui faut répercuter cette réponse, et l'heure logique qu'il faut accoler au message afin que le processus correspondant à la porte concernée puisse pointer dans son tableau local *TABLE*.

3.2. Le privilège d'émission de messages comb .

Dans chaque fragment, un seul processus est autorisé à émettre des messages **comb**. Au départ, tous les fragments sont réduits à un seul processus, et chacun d'eux est autorisé à émettre un message **comb**. Lors d'une fusion, le processus qui avait demandé la fusion, et était donc autorisé à émettre des messages **comb** perd cette autorisation dès lors qu'il reçoit un message **ok**, elle est supprimée. La réception d'un message **nok**, ou d'un message **cousin** implique que la porte par laquelle est parti le message **comb** ayant provoqué cette réponse ne sera plus utilisée pour envoyer un message **comb** (la gestion des portes est assurée par l'utilisation du tableau *CAND*.- voir § 5-) Si cela implique qu'il n'y a plus de porte disponible pour envoyer un nouveau message **comb**, le processus possédant l'habilitation la transmet à son père qui à son tour ferme la porte correspondante. Si c'est la racine qui possède l'habilitation et qu'elle ne dispose plus de porte, c'est que l'exécution de l'algorithme est terminée.

3.3. La terminaison.

Le mode de délivrance de l'autorisation à émettre le message **comb** correspond à un parcours d'arborescence dit recherche avec rebroussement (ou "backtrack"), sauf que, dans le cas présent, on ne peut parler de parcours en largeur ou en profondeur: le choix de la porte par laquelle envoyer le message **comb** se faisant aléatoirement, il peut s'agir aussi bien d'une porte donnant sur un fils que d'une porte donnant sur un autre processus (pas le père toutefois). Ainsi, lorsque c'est la racine d'un fragment qui possède le privilège, si elle n'a plus de porte disponible, l'exécution est terminée.

3.4. Le démarrage.

Pour ce qui est du démarrage de l'algorithme, on considérera que tous les processus sont susceptibles de démarrer ensemble, et que si un processus n'a pas démarré, il le fait dès lors qu'il reçoit un message.

4. Notations.

Les notations utilisées ici sont empruntées pour la plupart à la syntaxe de CSP. L'algorithme est spécifié de façon à en rendre la compréhension le plus facile possible. Il est bien entendu qu'une implémentation de cet algorithme sur un système distribué (Transputers par exemple) conduirait à un code syntaxiquement différent, mais proche (en OCCAM en particulier).

4.1 Le langage de spécification.

Dans la spécification de nos algorithmes, nous utilisons une syntaxe proche de celle de CSP. Toutefois, les modifications de syntaxe que nous apportons ont une répercussion importante sur la sémantique proprement dite. Ainsi nous considérons ici l'émission de message comme non bloquante; l' émission que nous noterons P_i!! <*x*>; signifiant envoi par la porte *i* du message <*x*>, de façon non bloquante. Dans le cas de processus d'identités toutes différentes, on peut considérer *i* comme étant l'identité du processus destinataire du message. Une telle hypothèse de non blocage implique que les processus soient munis d'un tampon servant de file d'attente pour les messages reçus. Pour que cette hypothèse de non blocage soit réaliste, il importe que la taille d'une telle file d'attente soit bornée, ce qui est le cas ici.

Symétriquement, nous noterons P_j ?? <*y*> la réception d'un message, avec la signification suivante:

- Le message est lu dans la file d'attente du processus courant, suivant la discipline propre à celui-ci (ici, Premier arrivé, premier servi, - PAPS-). La valeur de *j* est l'identificateur de la porte par laquelle est arrivé le message.

- La valeur transmise par le message est affectée à une variable locale au processus courant, de nom *y*. Les messages ne contiennent que des valeurs, aucun nom de variable.

Nous utilisons également quelques notations susceptibles d'alléger l'écriture, comme : $\cup_{x \in SON}$, P_x !!< *y* > signifiant alors, envoyer par toutes les portes dont les identificateurs sont dans le tableau *SON*, le message <*y*>

4.2 Les messages échangés.

Les messages sont composées de quatre champs, et notés <*a, b, c, d*>.

Entre l'émission et la réception d'un message, le passage se fait par liste de valeurs, et non par nom de variable. Ainsi, un message émis sous la forme :

- <*root*, **comb**, *open, h*> par un processus *y*, et reçu sous la forme:

- <*a, b, c, d*> par un processus *i* aura pour effet d'attribuer à la variable *a* du processus *i* la valeur de la variable *root* du processus *y*; à la variable *b* du processus *i*, la valeur **comb**; à *c* du processus *i* la valeur de *open* du processus *y*, et à *d* du processus *i* la valeur de *h* du processus *y*.

Le deuxième champ du message prend ses valeurs dans {**end, comb, ok, nok, equal, cousin, nrac, merge**}, c'est lui qui caractérise le type de traitement à effectuer par le processus récepteur.

Le premier champ est un identificateur de racine de fragment,

le troisième champ est une variable booléenne qui donne éventuellement l'état logique dans lequel il faut mettre la porte considérée.

le quatrième champ est une heure logique, il sert pour le chaînage de certains messages (cf. § 3.1.4).

Les premier, troisième, et quatrième champs ne sont pas toujours présents dans un message, le deuxième l'est toujours, c'est lui qui caractérise la nature du message.

4.3 Les variables et tableaux.

L'algorithme utilise des tableaux locaux tels que *CAND, SON* et *TABLE*, avec les significations suivantes:

- CAND[] est un tableau logique indexé par les identificateurs des portes du processus[1] et qui permet de tenir à jour les identificateurs des portes sur lesquelles ont peut encore éventuellement envoyer des messages "comb":

$$CAND[i] = \begin{cases} \text{vrai, signifie que la porte } i \text{ est utilisable} \\ \text{false, signifie que la porte } i \text{ n'est plus utilisable.} \end{cases}$$

Au départ, toutes les portes (logiques) du processus sont utilisables, donc en particulier *CAND[i]* = vrai pour tout *i*. Le passage à faux (ou éventuellement à vrai dans certains cas) de *CAND*[*i*] se fait dans les cas suivants:

- Lors de l'appel de la procédure ***UPDT***(*y, CAND, open*), l'élément indicé par *y* du tableau *CAND* est mis systématiquement à faux. Ceci se produit:

- lorsqu'un processus reçoit un message **ok**, celui-ci signifiant une acceptation de fusion, le processus contigu à la porte *y* par laquelle est arrivé le message **ok** est appelé à devenir le père du

[1]On suppose que chaque processus est capable localement de distinguer ses portes, c'est à dire qu'en fait il connaît l'existence de voisins différents sans avoir besoin pour autant de connaître l'identité de ceux-ci, ce qui lui permet éventuellement de les distinguer, même s'ils ont même identité.

processus courant dans le fragment en formation, par conséquent il n'est pas destiné à recevoir un message **comb**, et on doit poser *CAND*[*y*] := faux.

- lorsqu'un processus reçoit un message **nok** de l'un de ses fils, cela signifie que le fils en question était détenteur du privilège d'émission de message **comb**, et qu'il n'a plus de candidat possible (i.e. son tableau *CAND* est à faux partout) ce qui se marque par le fait que ce processus est tel que sa variable locale *open* a pour valeur faux (**∀y,** *open* := ∨ *CAND*[*y*] **)**. Le processus "ferme" alors la porte *x* par laquelle est arrivé le message **nok** (i.e. *CAND*[*x*] := faux).

- lorsque le processus courant reçoit un message **cousin**, il apprend ainsi que le processus qui lui envoie ce message fait partie du même fragment, et par conséquent il n'est plus utile de lui envoyer de message **comb**.

Lors de la réception d'un message fusion, ce message a été créé par la racine d'un fragment qui se fusionne dans un autre. Lors du transit de ce message, de la racine qui l'a émis à un processus du fragment dans lequel s'opère la fusion (le processus qui a répondu **ok**), il y a inversion de la relation père fils le long du chemin ainsi parcouru. La racine qui émet le message prend l'un de ses fils pour père, il lui faut donc recalculer la valeur de sa variable *open*, et la transmettre à son nouveau père afin qu'il tienne à jour la valeur correspondante de sa variable *CAND*[ ] qui peut, éventuellement, être remise à la valeur vrai alors qu'elle était précédemment à la valeur faux.

- *SON* est un tableau booléen indexé par les identités des portes et qui permet de connaître les fils du processus courant dans la construction de l'arbre.

{*SON*[*I*] = 1} ⇔ {Le processus contigu à la porte *i* est un fils}.

- *TABLE* est une structure de type liste dont les éléments sont indexés par les valeurs données à l'horloge logique locale (cf. § 3.1.4).

{*TABLE*[*h*] = (*e*,*k*)} ⇔ {A l'heure locale *h*, le processus a reçu un message par la porte *e* dont l'heure locale issue du processus contigu à la porte *e* était *k*.}.

- *i* est la variable locale dont la valeur est celle de l'identificateur du processus courant; cette valeur est un entier.

- *root* est une variable locale dont la valeur donne l'identificateur de la racine du fragment auquel appartient le processus courant.

- *father* est une variable locale dont la valeur donne l'adresse de la porte donnant sur le processus père dans l'arbre.

- *h* est une variable locale dont la valeur donne la dernière heure locale calculée.

- *crédit* est une variable locale dont la valeur donne le nombre de fois que le processus courant, s'il est racine d'un fragment, peut retirer une nouvelle valeur pour *i* (et par conséquent pour *root*).

h, crédit, father, root, i sont des variables à valeurs entières.

- *req* est une variable booléenne locale qui autorise une racine à émettre (si *req* = 1), ou non (*req* = 0), un message **comb**.

- *open* est une variable logique locale dont la valeur permet de savoir si un processus admet des candidats (i.e. des processus susceptibles de répondre **ok** à l'envoi d'un message **comb**).

- *ambiguity* est une variable logique locale dont la valeur permet de savoir qu'un processus a reçu une réponse **equal** à l'envoi d'un message **comb**, et qu'il est en attente de décision de la part de la racine du fragment dont il fait partie.

4.4. Les fonctions.

Afin d'alléger la spécification de l'algorithme, nous avons défini un certain nombre de fonctions qui sont:

SELECT qui est une fonction dont les paramètres sont une valeur x et un tableau *CAND*. ***SELECT*** choisit, aléatoirement, parmi les éléments du tableau *CAND* qui sont à la valeur vrai, et attribue la valeur d'indice, du tableau *CAND*, de l'élément ainsi trouvé à x.

Function ***SELECT*** (x,*CAND*, *a*, *b*) :: $x := 0$;
[*CAND*[1] = vrai $\wedge$ a$\neq$1 $\rightarrow x := 1$;
▮
CAND[2] = vrai $\wedge$ a$\neq$2 $\rightarrow x := 2$;
▮
... ...
▮
CAND[*n*] = vrai $\wedge\ a \neq n \rightarrow x := n$;
] ; [$x = 0 \wedge b \neq$ nil $\rightarrow x := b$] ; [$x = 0 \wedge b = i \rightarrow x := a$]

UPDT est une fonction de mise à jour de paramètres *y*, *CAND*, *open* qui permet de gérer *CAND* et *open*.

Procedure ***UPDT*** (*y*,*CAND*, *open*) ::
[*CAND*[*y*] := faux ;
open := $\vee$ *CAND*[*y*]
].

RANDRAW est une fonction qui effectue le tirage aléatoire sur [1,*d*], où $d = O(r.\lg r)$ d'une nouvelle identité pour un fragment, de valeur strictement supérieure à la précédente.

Procedure ***RANDRAW*** ::

$[\alpha :=$ ***random*** $(\{1,\ldots,d\})$; $i := i + \alpha$;

$\cup_{x \in SON}$ P_x !!<*i*,**nrac**,.,.> ; *req* := faux ; *root* := *i*].

La procédure ***INIT*** assure l'initialisation des variables.

Procedure ***INIT*** (*i, h, root,father, p, crédit , CAND, req, SON, open*) ::

[**Type** cellule = **record** index: integer;
heure: integer;
porte: integer;
suivant: cellule;
end
Type *TABLE* :**Liste of** cellule; **Type** *i, root,father, porte, crédit, h* : integer ;
Type*req, open, ambiguity* : boolean ; **Type***SON, CAND* : Array [1..*n*] **of** boolean ;
i:= ***random*** $(\{1,\ldots,d\})$; *root* := *i* ; *father*:= *i*; *h* := 0;*req* := faux ;

SON := ∅ ; *open*: = vrai ; $(\forall x \in NEIGHB)$ *CAND*[*x*] := vrai; *crédit* := *r* ; *ambiguity* := 0].

/* ici, $d = O(r.\lg r)$, où $r = 1/\varepsilon$) fixé */

La procédure ***TERM*** permet d'assurer la terminaison correcte de tous les processus.

Procedure ***TERM*** (*SON*) ::

[$\cup_{x \in SON}$ P_x !!<.,**end**,.,.> ; **STOP.**].

5. Spécification de l'algorithme

Proc ::
INIT ;
*[(*root* = *i* ∧ *req* = faux ∧ *ambiguity* := 0) →[*open* = *faux* → ***TERM***] ;
SELECT (*x*,*CAND*[ ],nil,nil) ; *h* := *h* +1; *req* := vrai ;
P_x !! <*i*,**comb**,.,*h*>
▮
P_y ??<*a,b,c,d*> →
[*b* = **end**→ ***TERM***
▮
b = **comb** ∧ *y* = *father* → [*open* = vrai → ***SELECT*** (*x*,*CAND*[ ],nil,nil) ; P_x!!<*a,b,c,d*>
▮
open = faux → P_y !!<.,**nok**,*open*,.>
]
▮
b = **comb** ∧ *y* ≠ *father* → [a = *root* → P_y!!<.,**equal**,.,.>
▮
a < *root* → P_y!!<*root*,**ok**,.,.>; *SON* := *SON* ∪ {*y*}
▮
a > *root* → P_y !!<.,**nok**,.,.>
]
▮
b = **ok** → [*y* ∈ *SON* → *SON* := {*SON* - {*y*}}∪{*father*};
[*root* = *i* → *root* := *a* ; ***UPDT*** ; P_y!!<.,**merge**,*open*,.>;
$\cup_{x \in SON}$ P_x !!<*a*,**nrac**,.,.>
▮
root ≠ *i* → *root*: := *a* ; P_{father} !!<*a,b*,.,.>
] ;*father* := *y*

```
        ]
    ▮
    b = nok → [c := faux → UPDT ] ;
                    [ root ≠ i → [ open = vrai ∧ ambiguity := 0→ SELECT (x,CAND[ ],y,father) ;
                                                                  P_x !!<root,comb,.,.> ;
                                    ▮
                                  open = faux→ P_father!!<a,nok,faux,.>
                                ]
                              ▮
                            root = i → req := faux
                          ]
    ▮
    b = equal → [ y ∈ SON → [i = root →[crédit > 0 → RANDRAW; crédit := crédit − 1;
                                                        ∪x∈ SON P_x !!<a,nrac,.,.>
                                    ▮
                                  crédit = 0 → P_y!!<.,cousin,.,d>
                                  ]
                              ▮
                            i ≠ root → h := h + 1 ; TABLE[h] = (y, e) ; P_father!!<a,b,c,h>
                          ]
                    ▮
                  y ∉ SON → ambiguity := 1 ; h := h + 1 ; TABLE[h] = (y, .); P_father!!<a,b,c,h>
                ]
    ▮
    b = cousin → [ambiguity = 1 → (t,.) := TABLE[e] ; UPDT(t,CAND,open) ;
                                  [open = faux → P_father!!<.,nok,faux,.,.>
                                    ▮
                                   open ≠ faux → SELECT (x,CAND[ ]) ; P_x!!<root,comb,.,.>
                                   ] ; ambiguity := 0
                    ▮
                  ambiguity = 0 → (k, l) := TABLE[d]; P_k!!<a,b,c,h>
                  ]
    ▮
    b = nrac → root = a ; ∪x∈ SON P_x !!<a, nrac,.,.> ;
                    [ambiguity = 1 → ambiguity := 0 ; SELECT (x,CAND[ ]); P_x!!<root,comb,.,.>]
    ▮
    b = merge → SON := SON ∪ {y} ; CAND[y] := c ; open := ∨k CAND[k] ;
                ∪x∈ SON P_x !!<a, nrac,.,.> ; [ root ≠ i → P_father!!<a,b,open,.> ]
  ]
].
```

6 Preuve de l'algorithme.

La preuve de cet algorithme repose sur l'invariant topologique de forêt, et sur le fait qu'il y a convergence probabiliste.

6.1 Le non-cyclage.

L'ensemble des identificateurs des fragments est un ensemble partiellement ordonné (voir [BIR-67]). C'est en utilisant la restriction de cette relation d'ordre large à l'ordre strict qu'on obtient

des fragments à structure d'arbres, c'est à dire sans cycle. Il nous faut montrer qu'il n'y a pas contradiction entre le caractère dynamique de l'algorithme, c'est à dire le possible retirage d'un numéro d'identité par l'un des fragments, et le respect de la relation d'ordre strict lors d'une fusion.

Soient E et F deux fragments d'identificateurs respectifs e et f. Dans un fragment, un seul processus peut envoyer une demande de retirage d'identificateur à la racine, c'est le processus autorisé à émettre un message **comb**. Supposons qu'on ait $e < f$, qu'un processus a de E ait envoyé un message **comb** à un processus, b, de F. Le processus b répond alors **ok** à a - ceci est dû au fait qu'on a $e < f$ -. Durant toute cette transaction, la valeur de e ne peut changer, puisque, a détenant le privilège d'émission de message **comb** dans E, il est le seul processus de E susceptible de demander un retirage d'identificateur. Un tel retirage ne sera demandé par a que sur réception du message **equal** en réponse à son message **comb**, message **equal** qui n'est ici pas émis puisque b teste que $e < f$ et envoie à a un message **ok**. La relation d'ordre ne peut donc être modifiée du côté de E. Par contre, il existe dans F un processus détenteur du privilège d'émission de message **comb** susceptible d'envoyer à la racine une demande de retirage d'identité, et ce processus est à priori différent de b. L'identificateur de racine de F peut donc à priori changer durant la transaction ci-dessus décrite. Soit f_1 le nouvel identificateur ainsi calculé, par construction, on a $f_1 > f$. Par transitivité $f_1 > e$, donc la transaction peut se poursuivre sans problème.

Au départ, tous les fragments sont réduits à un processus. On peut donc considérer que le graphe associé aux seuls fragments est une forêt couvrante. Notons Fusion(A éB) le fait que le fragment A fusionne dans le fragment B, on a:

$$\text{Fusion}(A \rightarrow B) \Rightarrow a < b.$$

De même, notons Tir (b) la fonction qui, à une racine b de fragment fait correspondre un nouvel identificateur b' Dans la spécification de l'algorithme ci-dessus, cette fonction correspond à la première ligne de la procédure ***RANDRAW*** (α := ***random*** ({1, ... ,d}) ; $b' := b + \alpha$) [1] α étant tiré de façon aléatoire sur un ensemble fini de nombres positifs non nuls, b étant positif et non nul, on a:

$$\forall \alpha \in [1,d] \; ; \; b + \alpha > b \text{ et donc }, \; b' = b + \alpha > b.$$

On a par conséquent:

$$\text{Fusion}(A \rightarrow \text{Tir}(b)) \Rightarrow a < b < b'.$$

6.2 Une forêt couvrante pour invariant.

[1] Afin de rendre la fonction compréhensible, on a remplacé i par b et introduit b' pour rester cohérent avec la définition de la fonction Tir.

Théorème 1: *L'ensemble des fragments constitue pour l'algorithme, un invariant de forêt couvrante.*

démonstration :

Par construction, on ne peut avoir Fusion(Tir (b)→ A). A chaque itération, le graphe des concaténations de fragments par des arêtes est sous graphe partiel de celui de la relation d'ordre sur les identificateurs des fragments, c'est par conséquent un sous graphe partiel d'un arbre, c'est à dire une forêt . En effet, le graphe associé à une relation d'ordre strict est celui d'un sup-demi treillis (voir un inf-demi treillis) -voir [BIR-67], c'est à dire en termes de théorie des graphes, un arbre dont la racine (i.e.le suprémum du treillis) est l'élément maximum dans la relation d'ordre A chaque fusion, deux fragments qui sont des arbres de la forêt sont au plus connectés par une arête. Si l'un des fragments contient p sommets, et l'autre q, ils contiennent respectivement $p-1$ et $q-1$ arêtes, et sont connexes chacun. Réunir ces deux fragments par une arête revient à former avec ceux ci une composante connexe qui contient alors $p+q$ sommets et $p-1 + q-1 +$ 1arêtes (le +1 étant dû à l'arête ajoutée pour opérer la concaténation) En posant $p + q = k$, on a constitué une composante connexe à k sommets et $k-1$ arêtes, c'est par conséquent un arbre. Au départ chacun des arbres de la forêt couvrante est réduit à un sommet. La forêt couvrante est donc bien un invariant de l'algorithme.

7 La convergence.

Théorème 2. *Dans tous les cas, ltout processus finit par savoir qu'il a terminé d'exécuter l'algorithme.*

démonstration :

L'étude de la convergence de l'algorithme doit se faire dans deux contextes différents, suivant que tous les identificateurs sont différents, ou s'il y a des égalités entre identificateurs.

7.1 Avec identificateurs différents

Supposons que la valeur de la variable *crédit* soit trés grande, par exemple *crédit* = r, où $r = 1/\varepsilon$, pour un $\varepsilon > 0$ donné. Chaque fois qu'un message **comb** sera envoyé sur une arête sortante (i.e. une arête ayant une extrémité dans un fragment, et l'autre extrémité dans un autre), les identités des fragments étant différentes, il y aura discrimination. Alors soit la fusion se fera si la relation d'ordre est respectée, et un message **ok** transitera dans l'arête en sens inverse du message **comb** (dans ce cas, une fusion s'ensuivra, et il y a donc convergence); soit la demande sera refusée, et ce sera un message **nok** qui transitera dans l'arête en sens inverse du message **comb**. Dans ce dernier cas, le processus recevant le message **nok** détient le privilège d'émission des messages **comb** dans

son propre fragment. Il choisit alors de façon aléatoire une porte *j* par laquelle envoyer un nouveau message **comb** pour laquelle *CAND*[*j*] a la valeur vrai et qui est différente de *y*, mais qui peut être son père. En particulier, dans l'éventualité où il n'y a pas d'autre porte possible, c'est le cas qui correspond à la commande gardée [$x = 0 \wedge b \neq$ nil $\rightarrow x := b$] de la procédure ***SELECT***.

Si on se trouve dans le cas où, étant racine, il n'y a pas d'autre porte possible que celle par laquelle est arrivé le **nok**, on renvoie le message **comb** par la même porte[1]. C'est la commande gardée [$x = 0 \wedge b =$ i $\rightarrow x := a$] de la procédure ***SELECT***. Dans ce cas, on enverra de tels messsages **comb** jusqu'à ce qu'arrive par la dite porte, soit un message **comb**, soit un message **ok** (ce qui signifie qu'alors l'autre fragment a changé d'identité), soit un message **equal** (ce qui signifie aussi qu'alors l'autre fragment a changé d'identité). Dans ce dernier cas on doit utiliser le traitement probabiliste des identificateurs égaux (voir alors § 7.2). Dans les cas autres que ce dernier, une fusion finit par s'opérer. En effet, si c'est un message **ok** qui revient, la fusion s'ensuit; si c'est un message **comb**, soit la relation d'ordre n'a pas changé, et alors la réponse à ce message **comb** est **ok**, soit la relation a changé de sens, et la prochain message **comb** qu'on enverra sur l'arête sera accepté. Une fusion finit d'autant plus par arriver que le nombre de fois qu'un fragment peut changer d'identité est fini C'est évident, car le changement d'identité d'un fragment ne peut être dû qu'à deux causes. Premièrement la fusion dans un autre, le nombre de processus étant fini, ce nombre de fusions est donc fini. Deuxièmement, le changement d'identité peut être dû à un retirage par la racine. Le nombre de retirages possibles est conditionné par la valeur de la variable *crédit*, il est donc aussi fini.

7.2 Avec des identificateurs égaux.

Lorsqu'il y a réception d'un message **equal** par un processus ayant émis un message **comb**, il s'agit de distinguer si les deux processus sont des cousins, c'est à dire s'il appartiennent à un même sous-arbre, ou si les deux processus appartiennent à des sous arbres différents mais dont les racines ont des identités égales. Pour opérer cette distinction, le fragment qui reçoit la réponse **equal** (et celui-là seulement) tire une autre identité, qui est plus grande. La racine envoie alors un message de mise à jour **nrac** à tous ses fils, et celui qui possède le privilège d'émission de message **comb** peut alors renvoyer un tel message. Si la racine ne peut plus changer d'identité (*crédit* = 0), alors elle renvoie au processus possédant le privilège, et à lui seul, un message **cousin** qui lui fait alors considérer le processus ayant répondu **equal** comme appartenant au même fragment.

Ainsi, dans un fragment dont la racine a un crédit nul, toute demande de retirage de l'identificateur provoque une émission de message **cousin** vers le processus détenteur du privilège d'émission des messages **comb**. La réception dudit message **cousin** par le processus considéré provoque chez celui-ci la mise à la valeur faux d'un élément du tableau *CAND*. Ce tableau étant de

[1]On pourrait imaginer une version de l'algorithme dans laquelle on introduirait une attente sur la porte considérée dans ce cas là, ce qui serait susceptible de diminuer le nombre de messages échangés dans le pire des cas.

taille fixe,la variable *open* du processus finit par prendre la valeur faux, le processus finit donc par transmettre le privilège à son père. Le nombre de processus étant fini, le privilège finit par revenir à la racine, dont la variable *open* prend finalement aussi la valeur faux sans que pour autant la variable ambiguïty ait la valeur 1 (le privilège étant transmis par un fils).

On est alors dans le cas où, open étant à la valeur faux, on a:

$root = i \wedge req = \text{faux} \wedge ambiguity := 0$

ce qui entraîne la terminaison propre de l'algorithme par réception par tout processus du fragment d'un message de terminaison.

On considérera que si un message provenant d'un processus n'ayant pas encore reçu de message de terminaison, du même fragment, ou d'un autre arrive sur un processus qui a terminé (i.e. qui a reçu un message **end**), il est retourné tel quel à l'envoyeur, ce qui finit aussi par entraîner une terminaison propre.

On pourrait, lors de la terminaison d'un processus lui faire envoyer un message à tous les processus encore candidats, un message provoquant la fin de toute communication entre eux, ce qui nécessiterait éventuellement une mise à jour des tableaux *CAND* des processus concernés, il s'agit là d'une variante dont la prise en compte était de nature à alourdir encore la spécification de l'algorithme, mais qui pourrait être intéressante lors d'une implémentation réelle.

8. La terminaison.

8.1 Terminaison probabiliste.

Théorème 3. *A la terminaison de l'algorithme, la probabilité qu'un arbre couvrant ait été construit est* $\geq 1 - \varepsilon$.

démonstration :

La terminaison s'obtient dés lors qu'une racine a épuisé son crédit, tous les messages **equal** qui seront reçus sur toutes les arêtes sortantes du fragment provoquerons le passage à la valeur faux de la variable *CAND*[] correspondante, de même que les réponses **ok**. A terme, s'il n'y a eu sur aucune arête sortante de réponse **nok**, toutes les variables *open* de tous les processus du fragment vont prendre la valeur faux, et la racine du fragment considère alors l'algorithme comme terminé.

Soient donc ε la probabilité d'erreur de l'algorithme et $d \geq 1/\varepsilon$. La probabilité que deux identités tirées sur I de taille d est donc $1/d$. Comme la variable *crédit* est initialisée à $r = 1/\varepsilon$, la probabilité que l'algorithme échoue dans la construction d'un arbre couvrant est majorée par la quantité $\sum_{i=1}^{r} (\frac{1}{d})^i < \frac{1}{d-1} < \varepsilon$.

8.2 Terminaison déterministe.

Une autre façon d'obtenir une bonne terminaison consiste, si on connaît n, nombre de processus de l'algorithme[1], à établir alors un comptage des processus de chaque fragment au fur et à mesure de la formation de ceux-ci, et dès lors qu'un fragment contient n processus, on sait, de façon sûre que l'algorithme est terminé. On n'est donc pas obligé ici de gérer une variable *crédit*, mais on ne peut alors prédire exactement le nombre de retirages nécessaires à la terminaison. Dans ce cas, il suffit de prendre un domaine de tirage I de taille d tel que $d \geq n^2/\varepsilon$, pour que la probabilité d'avoir deux identités quelconques égales lors d'un tirage aléatoire soit inférieure à ε, $0 < \varepsilon < 1$ fixé. Voir la démonstration de cette propriété en Annexe1.

De même, si $N < n \leq kN$, il suffit de prendre $d \geq (1 - 1/k)^2 \times N^2/\varepsilon$, ou bien $d \geq N^2/\varepsilon$ si seul un minorant N de n est connu ; voir la démonstration en Annexe 2.

On trouvera ci-dessous la spécification de l'algorithme lorsque n est connu.

9. L'algorithme avec connaissance de n.

Proc ::
INIT ; *count* := 1;
*[(*root* = *i* ∧ *req* = faux ∧ *ambiguity* := 0) →[*open* = *faux* → ***TERM***] ;
SELECT (*x*,*CAND*[ ]) ; *h* := *h* +1; *req* := vrai ; P_x !! <*i*, **comb**,*count* ,., *h*>;
▮
P_y ??<*a*,*b*,*c*,*d*, *e*> →
[*b* = **end**→ ***TERM***
▮
b = **comb** ∧ *y* = *father* → [*open* = vrai → ***SELECT*** (*x*,*CAND*[ ]) ; P_x!!<*a*, *b*, *c*, *d*, *e*>
▮
open = faux → P_y !!<.,**nok**,.,*open*, >
]
▮
b = **comb** ∧ *y* ≠*father* →[a = *root* → P_y!!<., **equal**,.,.,.,>
▮
a < root → P_y!!<*root*, **ok**,., , >
▮
a > *root* → P_y !!<.,**nok**,.,.,.,>
]
▮
b = **ok** → [*y* ∈*SON* → *SON* := {*SON* - {*y*}}∪{*father*};
[*root* = *i* → *root* := *a* ;***UPDT*** ; P_y!!<.,**merge**,*count*, *open*,.>;
∪*x*∈ *SON* P_x !!<*a*, **nrac**,.,.,.,.>
▮
root ≠ *i* → *root*: := *a* ; P_{father} !!<*a*, *b*,.,.,.>
];*father* := *y*
]
▮
b = **nok** → [*c* := faux → ***UPDT***];
[*root* ≠ *i* → [*open* = vrai ∧ *ambiguity* := 0 → ***SELECT*** (*x*,*CAND*[ ]) ;
P_x !!<*root*,**comb**,.,., > ;

[1]En fait, il suffit qu'un seul processus connaisse n, dans ce cas, il faut prévoir un champ de plus au message, et passer cette valeur au fur et à mesure des fusions.

```
                                                    ▮
                                                open = faux→ P_father!!<a, nok,., faux,.>
                                                ]
                                        ▮
                                root = i  → req := faux
                                ]
        ▮
b = equal →[ y ∈ SON  →[i = root  →RANDRAW; ∪x∈ SON Px !!<a, nrac,.,.,.>
                                        ▮
                                  i ≠ root  → h := h + 1; TABLE[h] := [y, e); P_father!!<a, b, c, d, h>
                                  ]
                        ▮
                y ∉ SON  → ambiguity := 1 ; h := h + 1 ; TABLE[h] := (y, .); P_father!!<a, b, d, c, h>
                ]
        ▮
b = cousin → [ambiguity = 1 → (t,.) := TABLE[e]; UPDT(t,CAND,open);
                                        [open = faux → P_father!!<.,nok,., faux,.>
                                            ▮
                                         open ≠ faux → SELECT (x,CAND[ ]) ; Px!!<root,comb,.,.,.>
                                         ] ; ambiguity := 0
                        ▮
                ambiguity = 0 → (k, l) := TABLE[d]; Pk!!<a, b, c, d, l>
                ]
        ▮
b = nrac → root = a ; ∪x∈ SON Px !!<a, nrac,.,.,.>;
                        [ambiguity = 1 → ambiguity := 0 ; SELECT (x,CAND[ ]) ; Px!!<root,comb,.,.,.>]
        ▮
b = merge → SON := SON ∪ {y} ; CAND[y] := d ; open := ∨k CAND[k] ;
                ∪x∈ SON Px !!<a, nrac,.,.,.>;
                        [ root ≠ i  → P_father!!<a, b, c, open,.>
                            ▮
                         root = i  → count := count + c ; [ count = n → TERM ]
                         ]
    ]
].
```

* *

Bibliographie

[AF,SA-87] **Y. AFEK & M. SAKS**, Detecting global termination conditions in the face of uncertainty, *Proc. 6th ACM PODC*, Vancouver, B.C., 1987, pp. 109-124.

[ANG-80] **D. ANGLUIN**, Local and global properties in networks of processors, *Proc. 12th Annual ACM Symposium on Theory of Computing* (Los Angeles, Calif., April 1980). ACM, New York 1980, pp.82-93.

[AT,SN,WA-88] **H. ATTIYA, M.SNIR** and **M. K. WARMUTH**, Computing on an Anonymous Ring, *J. ACM*, Vol. 35, N°4, Oct. 1988, pp.845-875.

[AWE-87] **B. AWERBUCH,** Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election and related problems, Proc. 19th *ACM-STOC*, pp. 230-240, 1987.

[BIR-67] **G. BIRKOFF**, Lattice theory , *American Mathematical Society,* Vol. XXV,1967.

[CI,JA,SI-87] **I. CIDON, J. JAFFE & M. SIDI**, Local Distributed Deadlock Detection by Cycle detection and Clustering, *IEEE Trans.on Software Engineering*, Vol.SE13, N° 1, January 1987, 3-14

[GA,HU,SP-85] **R.G. GALLAGER, P.A. HUMBLET, & P.M. SPIRA**, A Distributed algorithm for minimum weight spanning trees, *ACM Trans. Prog.and Syst.* **5**, 1985, pp. 66-77.

[HOA-78] **HOARE C.A.R.**,*Communicating sequential Processes*, Comm.of ACM, Vol. 21 n°8 August 1978, pp. 666-677.

[LAV-88] **I. LAVALLÉE**, A fully distributed asynchronous algorithm for leader election and (minimum) spanning tree problems, *Symposium ICOMID*, Ho Chi Minh ville, avril 1988..

[LA,LA-89] **I. LAVALLÉE & C. LAVAULT**, Yet another distributed election and spanning tree algorithm RR-INRIA n°1024 Avril 1989.

[LA,RO-86] **I. LAVALLÉE & G. ROUCAIROL**, A Fully distributed (minimal) spanning tree algorithm, *Information Processing Letters* **23**, 1986, pp.55-62.

[LEL-77] **G. LELANN**, Distributed System. Towards a Formal Approach. IFIP Congress 1977, North Holland, 1977, 155-160.

[MA,AF-89] **Y.MATIAS & Y AFEK**,Simple and efficient election algorithms for anonymous networks, Proc. 3rd IWDA' 89, Nice, sept 89, J.C. Bermond et M.Raynal eds, LNCS n°392, pp. 183, 194.

[MAT-87] **F. MATTERN**, Algorithms for distributed termination detection, *Distributed Computing*, Vol.2, n° 3, 1987, pp.161-175.

[SOL-61] **M. SOLLIN**, 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.

[YAM-88] **M. YAMASHITA** and **T. KAMEDA**, Computing on anonymous networks, *Proc. 7th ACM Symp. PODC*, Toronto 1988, pp.117-131.

*

* *

ANNEXE 1

Evaluation de la taille *d* du domaine *I* lorsque *n* est connu

On effectue une série de n tirages aléatoires dans $I = [1,d]$. Soient id_i et id_j deux identités de deux processus quelconques P_i et P_j tirées au hasard dans cette serie.

Soit $p = \Pr\{id_i \neq id_j\}$. On a alors,

$$p = \frac{d\ (d-1)\ \ldots\ (d-n+1)}{d} = \prod_{i=0}^{n-1} (1-\frac{i}{d}).$$

Comme pour tout réel $x \leq \frac{1}{2}$, $1 - x > e^{-2x}$, il suffit de faire l'hypothèse que $n/d \leq \frac{1}{2}$, c'est à dire $d \geq 2n$, pour avoir les inégalités suivantes

$$p = \prod_{i=0}^{n-1} (1-\frac{i}{d}) > \prod_{i=0}^{n-1} e^{-2i/d} > e^{-n^2/d}.$$

Comme $(\forall\ 0 < \varepsilon < 1)\ \varepsilon > \ln(1-\varepsilon)$, $p > 1 - \varepsilon$ dès que $d \geq n^2/\varepsilon$. Par conséquent, les identités des n processus doivent être tirées aléatoirement d'un domaine I de taille $d \geq n^2/\varepsilon$, pour un ε donné ($0<\varepsilon<1$). Dans ce cas, toutes les identités tirées seront distinctes avec une probabilité $> 1 - \varepsilon$.

*

* *

ANNEXE 2

Evaluation de la taille d du domaine I lorsque N est connu, $N < n \leq kN$

Le calcul est exactement semblable à l'évaluation effectuée dans l'Annexe 1.

i) Soit on effectue une série de N tirages aléatoires dans $I = [1,d]$. Soient id_i et id_j deux identités de deux processus quelconques P_i et P_j tirées au hasard dans cette serie.

Soit $p = \Pr\{id_i \neq id_j\}$. On a alors,

$$p = \frac{d\ (d-1)\ \ldots\ (d-N+1)}{d} = \prod_{i=0}^{N-1} (1-\frac{i}{d}).$$

Comme pour tout réel $x \leq \frac{1}{2}$, $1 - x > e^{-2x}$, il suffit de faire l'hypothèse que $N/d \leq \frac{1}{2}$, c'est à dire $d \geq 2N$, pour avoir les inégalités suivantes

$$p = \prod_{i=0}^{N-1} (1-\frac{i}{d}) > \prod_{i=0}^{N-1} e^{-2i/d} > e^{-N^2/d}.$$

Comme $(\forall\ 0 < \varepsilon < 1)\ \ \varepsilon > \ln(1-\varepsilon)$, $p > 1 - \varepsilon$ dès que $d \geq N^2/\varepsilon$. Par conséquent, les identités des n processus doivent être tirées aléatoirement d'un domaine $\boldsymbol{I}$ de taille $d \geq N^2/\varepsilon$, pour un ε donné ($0<\varepsilon<1$). Dans ce cas, toutes les identités tirées seront distinctes avec une probabilité $> 1 - \varepsilon$.

ii) Soit on veut également prendre en compte le majorant kN. Alors, comme $(k-1)N \leq kN$, on a $N \geq (\frac{k-1}{k})N$, et les calculs se mènent de la même manière.

Les identités des processus doivent être tirées aléatoirement d'un domaine $\boldsymbol{I}$ de taille $d \geq (1 - \frac{1}{k})^2 N^2/\varepsilon$, pour un ε donné ($0<\varepsilon<1$). Dans ce cas, toutes les identités tirées seront distinctes avec une probabilité $> 1 - \varepsilon$.

Exemple : Si $k = 2$, $d \geq N^2/4\varepsilon$; si $k = 3$, $d \geq \frac{4}{9}N^2/\varepsilon$.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique