Détecter la perte de jetons et les regénerer sur une structure en anneau

Michel Raynal, Gerardo Rubino

INRIA

CENTRE DE RENNES
IRISA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél (3) 954 90 20

Rapports de Recherche

N° 428

DÉTECTER LA PERTE DE JETONS ET LES RÉGÉNÉRER SUR UNE STRUCTURE EN ANNEAU

Michel RAYNAL
Gerardo RUBINO

Juillet 1985

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

Publication Interne n° 261
Juillet 1985
24 pages

DETECTER LA PERTE DE JETONS ET LES REGENERER SUR UNE STRUCTURE EN ANNEAU

Michel RAYNAL* , Gerardo RUBINO**

* IRISA/Université de Rennes I
** IRISA/INRIA Rennes

Campus de Beaulieu
35042 Rennes-Cedex
FRANCE

Résumé : De nombreux systèmes distribués utilisent des structures en anneau sur lesquelles circulent des messages spéciaux (les jetons) réalisant le contrôle des applications. La fiabilité des algorithmes de contrôle distribué nécessite alors de savoir détecter la perte des jetons et, si tel est le cas, de les regénérer. C'est à ce problème qu'est consacré ce rapport. Un nouvel algorithme basé sur l'existence de plusieurs jetons qui détectent mutuellement leurs pertes respectives est présenté. Son originalité est, contrairement aux algorithmes "classiques", de n'utiliser ni horloge de garde, ni identité des sites qui constituent le système distribué ; il est fondé sur le principe des compteurs monotones croissants.

Abstract : Numerous distributed systems use ring structures with special messages (tokens) to control applications. In order to have reliable distributed control algorithms, it is necessary to detect the loss of the tokens and if that is the case, to regenerate them. This report deals with this problem. A new algorithm is presented for solving it. It is based on the existence of several tokens detecting their mutual loss. Unlike "classical" algorithms its originality lies in using neither time-out nor process identity but monotonically increasing counters.

1. INTRODUCTION.

Dans le domaine de l'algorithmique distribuée de nombreux algorithmes, dont le but est de réaliser un contrôle du système réparti dans lequel ils apparaissent, sont basés sur l'existence d'un message spécial appelé **jeton**. Lorsque le jeton est unique, sa possesion par un processus assure à ce dernier une exclusivité qui peut être mise à profit pour réaliser une discipline de contrôle particulière; citons à titre d'exemples la réalisation d'une exclusion mutuelle [LEL77, RIC83], le droit de modifier une copie de fichier dupliqué [LEL78] et la réalisation d'une élection [CHA79, DOL82].

Les algorithmes basés sur l'existence d'un jeton ont souvent l'advantage d'être simples. Cette simplicité est réelle lorsque les lignes de communication sont fiables, c'est-à-dire lorsqu'il n'y a ni altérations ni pertes de messages. Il n'en est plus de même lorsque les lignes ne sont pas fiables car la perte du jeton est alors fatale à l'algorithme distribué qui ne peut plus progresser. Deux problèmes se posent alors au concepteur de l'algorithme : détecter la perte du jeton et, si tel est le cas, le regénérer.

La perte du jeton peut a priori être détectée par un ou plusieurs processus ; dans tous les cas il ne faut en regénérer qu'un exemplaire unique. Plusieurs solutions ont été proposées pour résoudre ces problèmes ; la plupart s'appuient sur une topologie de communication entre processus particulière et sur l'utilisation des mêmes méthodes. Considérons d'abord l'aspect structurel des lignes. Il n'y a pas de processus privilégié vis a vis du jeton ; en conséquence la topologie des communications est non-hierarchique et tout processus doit pouvoir communiquer (directement ou non) avec tout autre processus ; les structures considérées, dans les algorithmes basés sur les jetons, sont donc l'anneau et le maillage complet. Pour la détection de la perte, la méthode couramment utilisée est l'emploi d'horloges de garde [LEL77]. Lorsqu' un processus transmet le jeton à son successeur sur l'anneau par exemple, il arme une horloge de garde. Le calibrage du délai correspond au temps au bout duquel ce processus devrait avoir à nouveau reçu le jeton. Lorsque le délai est écoulé le processus entame une négociation avec les autres processus placés sur l'anneau afin de determiner si le jeton est effectivement perdu (cette phase peut être recommencée plusieurs fois si les messages relatifs à la négociation se perdent). Il est crucial de ne regénérer le jeton que s'il est perdu et de ne le regénérer alors qu'en un seul exemplaire ; pour

assurer cette unicité les identités des processus (qui doivent être toutes distinctes) sont utilisées.

Les algorithmes de détection de la perte de jeton et de regénération sont ainsi généralement complexes : utilisant l'identité des processus ils présentent un caractère dissymétrique (les processus n'ont le même texte qu'à l'identité prés et en conséquence ont des comportements différents) et la panne des processus peut à nouveau leur être fatale.

Ce rapport présente des solutions au problème de la perte d'un jeton (détection et regénération) qui ne souffrent pas des inconvénients précédents, puisqu'ils n'utilisent ni horloge de garde ni identité des processus. La Section 2 présente un algorithme dû à Misra qui utilise deux jetons. Dans la Section 3 nous présentons un nouvel algorithme distribué et symétrique à deux jetons fondé sur un principe différent que nous généralisons à k jetons dans la Section 4 pour améliorer ses performances. Le principe sur lequel est fondé l'algorithme consiste à utiliser des compteurs monotones croissants.

2. L'ALGORITHME DE MISRA.

Hypothèses.

Misra a proposé dans [MIS83] un algorithme de détection de perte de jeton et de regénération qui n'utilise ni horloge de garde, ni identité des processus. Avant de l'énoncer rappelons les hypothèses contextuelles sur lesquelles il s'appuie (ces hypothèses sont valables pour tous les algorithmes présentés dans ce rapport). Les processus sont situés sur un anneau uni-directionnel (du point de vue du transfert du jeton et donc du privilège associé). Les lignes de communication sur cet anneau peuvent perdre les messages mais ne les altèrent pas. De plus les messages ne peuvent pas se doubler sur une ligne de communication (mais le peuvent à l'intérieur d'un processus).

Principe.

La solution proposée par Misra repose sur un principe simple : elle consiste à utiliser deux jetons dont chacun sert à détecter la perte éventuelle de l'autre. Le procédé adopté est le suivant : un jeton arrivé à P_i peut garantir que l'autre jeton est perdu (et alors le regénérer) si depuis le passage précédent de ce jeton à P_i, ni lui ni le processus P_i n'ont rencontré l'autre jeton. Les deux jetons ont des comportements symétriques, chacun permettant de détecter la perte de l'autre (du point de vue du privilège associé au jeton, un seul d'entre eux est pertinent). La perte d'un jeton est détectée par l'autre en un tour sur l'anneau. Cet algorithme ne fonctionne donc que si, un jeton étant perdu, l'autre effectue un tour complet sans se perdre. Nous proposons dans la Section 4 un algorithme utilisant un nombre quelconque de jetons ; dans ce cas, l'algorithme fonctionne tant qu'il reste au moins un jeton sur l'anneau. Dans chaque contexte particulier on peut a priori, par un choix conséquent du nombre de jetons, rendre la probabilité de défaillance de l'algorithme aussi faible que l'on veut (Rappelons que les algorithmes "classiques" ne sont pas exempts de ce problème, les messages de négociation entre processus pouvant être indéfiniment perdus).

L'algorithme.

Soient j_0 et j_1 les deux jetons. Leurs passages dans chaque processus et leurs rencontres devant être mémorisées, on les value par un entier qui est transporté par les jetons : vj_0 et vj_1 respectivement ; la valeur absolue de chacun d'eux compte le nombre de fois où les jetons se sont rencontrés, ces

deux nombres étant liés par l'invariant :

$$vi_0 + vi_1 = 0 \ .$$

Initialement les jetons sont dans un processus quelconque de l'anneau et l'on a $vi_0 = 1$ et $vi_1 = -1$. Chaque processus P_i est doté d'une variable entière m qui mémorise la valeur associée au dernier jeton qu'a vu passer le processus :

var m : **entier initialisé à** 0.

Les processus (sites) étant les seules entités où peuvent être effectués des calculs, l'algorithme est décrit par les évenements (et les traitements associés) dont les processus sont les sièges. Le comportement de chaque processus P_i est le suivant :

lors de

la réception de (i_0, vi_0)
faire
si $m = vi_0$ **alors**
début
(* i_1 est perdu : le regénérer *)
$vi_0 := vi_0 + 1$;
$vi_1 := -vi_0$
fin
sinon $m := vi_0$
fsi
fait

la réception de (i_1, vi_1)
faire
(* traitement analogue au précédent en
intervertissant les rôles de i_0 et i_1 *)
fait

la rencontre de i_0 **et** i_1
faire
$vi_0 := vi_0 + 1$;
$vi_1 := vi_1 - 1$
fait

Commentaires.

L'algorithme consiste à maintenir toujours vraie la relation $vi_0 + vi_1 = 0$. Pour cela lorsque les deux jetons se rencontrent (rappelons qu'ils ne peuvent se rencontrer que dans un processus) on modifie les valeurs qui leur sont associées en conséquence. Lorsque P_i reçoit l'un des jetons, il teste si sa variable m a la valeur du jeton reçu, soit vi_0. Si $m \neq vi_0$, le jeton j_1 soit est nécessairement passé par P_i, soit a rencontré j_0 : il n'est donc pas perdu : P_i mémorise alors le passage de j_0 par l'instruction $m := vi_0$. Dans le cas où à la réception de j_0 le site P_i trouve m égal à vi_0, d'une part le jeton j_1 n'est pas passé par P_i (sinon m aurait été modifié en conséquence) et d'autre part, les deux jetons ne se sont pas rencontrés sur l'anneau (sinon le compteur vi_0 aurait une valeur supérieure à son ancienne valeur) ; en conséquence le jeton j_1 est perdu. Le processus P_i le regénére en affectant aux deux compteurs vi_0 et vi_1 les valeurs adéquates.

Borner le domaine des compteurs.

La taille des compteurs vi_0 et vi_1 associés aux jetons n'est pas bornée a priori, ce qui peut constituer un inconvénient pour l'implémentation de cet algorithme. Nous allons voir que les propriétés de l'algorithme permettent de borner ces compteurs.

L'algorithme n'utilise en effet que des tests de comparaisons sur les entiers, du type égalité/inégalité et non du type inférieur/supérieur (ce qui de plus en autorise une mise en oeuvre matérielle simple). Lorsque les compteurs sont mis à jour (lors d'une rencontre ou d'une regénération) ils sont incrementés (en valeur absolue) et prennent alors des valeurs différentes de celles prises par les variables m_i des processus P_1 à P_n. Il est donc possible d'incrémenter les compteurs vi_0 et vi_1 modulo $n+1$; vi_0 par exemple, ne pourra pas alors prendre la même valeur que l'une des variables m_i : pour que cela se produise il faudrait que vi_0 aitt été incrementé $n+1$ fois depuis sa dernière visite à P_i ce qui est impossible puisqu'il n'y a que n processus et que les jetons ne se rencontrent que dans les processus et s'y rencontrent au maximum une fois.

Compteurs à valeurs booléennes.

Dans l'algorithme de Misra que nous venons de présenter l'identité des processus n'intervient pas : ceux-ci peuvent être placés de manière quelconque sur l'anneau, ce qui constitue une propriété intéressante, aucun processus ne

jouant un rôle privilégié. Toutefois si l'identité des processus n'intervient pas il n'en est pas de même de leur nombre *n* lorsque l'on borne le domaine de définition des compteurs : ce dernier est fonction du nombre de processus placés sur l'anneau. Est-il possible d'éliminer cette contrainte ?

Considérons l'algorithme précédent et prenons comme compteurs des booléens. Chacun des compteurs est respectivement initialisé à *vrai* et *faux* (les variables *m* étant initialisées à la valeur *nil*). La relation à conserver invariante entre les compteurs est donc vi_0 = **non** vi_1 : lors d'une rencontre les jetons échangent leurs valeurs de façon à mémoriser cette rencontre. Pour voir si l'algorithme est correct, plaçons-nous du point de vue d'un processus P_i quelconque sur l'anneau : si les jetons ne se perdent pas, P_i doit voir passer la séquence de valeurs *(vrai ; faux)** indépendamment des jetons qui véhiculent ces valeurs. Pour qu'il en soit ainsi il est nécessaire que lorsque les jetons se rencontrent dans un processus, le dernier jeton arrivé double l'autre jeton. En effet ceci assure que les valeurs *vrai / faux* véhiculées par les jetons seront vues par chaque processus selon la séquence *(vrai ; faux)**. Si par contre lors d'une rencontre les jetons peuvent ou non se doubler l'algorithme devient incorrect, des regénérations de jetons non perdus pouvant se produire.

Conclusion

En conclusion rappelons les principales propriétés de l'algorithme de Misra : dans une structure de processus connectés en anneau, la détection de la perte d'un jeton est effectuée sans utiliser ni horloge de garde ni identité des processus, seul le nombre de processus intervient lorsque l'on borne le domaine de définition des compteurs manipulés par l'algorithme. Il est possible d'éliminer cette dépendence et de ne considérer que des valeurs booléennes pour les jetons mais lors de leurs rencontres ils doivent alors obligatoirement se dépasser ; ceci peut par exemple correspondre à une gestion LIFOP (Last In First Out with Preemption, i.e. Dernier Arrivé Premier Sorti avec Préemption) des jetons dans chaque site.

3. UN ALGORITHME A 2 JETONS BASE SUR LE COMPTAGE.

Nous proposons un algorithme à 2 jetons fondé sur un principe différent de celui de Misra que nous généraliserons à un nombre quelconque de jetons dans la Section 4. Comme précédemment les jetons tournent dans le même sens sur l'anneau des processus : ..., P_i, P_{i+1}, ... (les opérations sur les indices des processus sont réalisées modulo n).

Principe.

L'algorithme est basé sur l'idée suivante : lorsqu'un jeton h passe dans un site P_i il mémorise son passage dans une variable locale de P_i en lui affectant la valeur d'un compteur qu'il transporte, après avoir incrémenté ce compteur ; celui-ci précise à tout instant le nombre de sites visités par ce jeton. La trace laissée par un jeton, lors de son passage en P_i, permettra à l'autre jeton de détecter sa perte éventuelle lorsqu'il arrivera en P_{i+1}.

La détection de la perte du jeton h entre les sites P_i et P_{i+1} par l'autre jeton j se fait de la manière suivante. Outre son compteur le jeton j véhicule la valeur de la trace qu'a laissée le jeton h lors de son passage en P_i. Lorsque j arrive en P_{i+1}, il doit trouver la trace qu'y a laissé le jeton h égale à la trace que h a laissé en P_i (et que j transporte) augmentée de un (puisque h a visité un site de plus) ; si tel n'est pas le cas, le jeton h s'est perdu entre P_i et P_{i+1}, la détection de la perte ayant donc lieu en P_{i+1}.

Pour que le principe soit correct il reste à examiner le comportement que doivent présenter les deux jetons lorsqu'ils se rencontrent car ils peuvent alors éventuellement se doubler. Si tel est le cas, soit j double le jeton h, il va trouver une rupture de séquence dans les traces laissées par h dans les processus ; celle-ci ne doit pas l'amener à conclure à la perte de h. Pour cela, chacun des jetons est doté d'un drapeau s_j qu'il met à vrai lorsqu'il rencontre l'autre jeton. En conséquence, lorsque j arrive dans un processus il se comportera comme nous l'avons vu si son drapeau est à *faux* ; si par contre s_j est à *vrai* le jeton j met à jour l'image de la trace de h qu'il véhicule et repositionne son drapeau à *faux*. L'introduction des drapeaux est donc dûe au fait que les jetons ont des vitesses de rotation indépendantes et a priori différentes dans l'anneau des processus et lorqu'ils se rencontrent dans l'un des sites ils peuvent s'y doubler.

L'algorithme.

Chaque jeton se présente comme un triplet (j, v_j, s_j) dans lequel :

* j désigne son identité ($j = 0,1$),
* v_j est un tableau à 2 entrées :
  - $v_j[j]$ compte le nombre de processus visités par j.
  - $v_j[h]$ mémorise la dernière valeur de la trace laissée par le jeton h ($h = j + 1$ **mod** 2) dans le dernier processus visité par j.
* s_j est le booléen mis à *vrai* lors d'une rencontre.

Chaque processus P_i est doté d'un tableau d'entiers m à deux entrées correspondant aux deux jetons j et h ; ce tableau permet à chacun des jetons de laisser la trace de son passage dans le site.

Au début les jetons sont placés de façon quelconque dans l'anneau des processus : les champs v_j et s_j des jetons et les variables m_i des processus sont initialisés en conséquence (voir la Section 4 où les initialisations sont décrites dans le cas général d'un nombre quelconque de jetons).

Le comportement d'un processus P_i quelconque est décrit par l'algorithme suivant :

(page suivante)

```
lors de la réception de ( j, v_j, s_j )
    faire
        v_j[j] := v_j[j] + 1;
        m[j] := v_j[j];
        h := j + 1 mod 2;
        si h présent alors
            début «rencontre des deux jetons»
                v_j[h] := m[h]; s_j := vrai;
                v_h[j] := m[j]; s_h := vrai
            fin
(α)     sinonsi m[h] ≠ v_j[h] + 1 et non s_j alors
            début «regénération du jeton h»
                v_h := v_j;
                «simulation arrivée de h»
                v_h[h] := v_h[h] + 1;
                m[h] := v_h[h];
                «simulation de la rencontre»
                v_j[h] := m[h]; s_j := vrai;
                v_h[j] := m[j]; s_h := vrai
            fin
            sinon
                début
(β)                 v_j[h] := m[h]; s_j := faux
                fin
            fsi
        fsi
    fait
```

Commentaires.

La détection de la perte d'un jeton (ici noté h) a lieu à la ligne α : le drapeau s_j est à *faux* et il y a rupture de séquence dans la trace de h. Le jeton perdu est regénéré et les contextes des jetons et du processus dans lequel la regénération a lieu sont mis à jour (simulation de l'arrivée du jeton regénéré et de la rencontre entre les deux jetons). S'il n'y a pas rupture de séquence où s'il y a eu possibilité de dépassement des jetons il n'y a pas détection de (fausse) perte mais seulement des mises à jour (ligne β). Les mises à jour redondantes sont laissées par souci de clarté dans l'éxposé de

l'algorithme.

Borner le domaine des compteurs.

De la même façon que dans le cas de l'algorithme de Misra, on peut faire évoluer les compteurs associés aux deux jetons (i.e. $v_j[j]$ et $v_h[h]$) modulo $n+1$ en évitant ainsi les problèmes d'implémentation associés à l'absence de bornes.

Remarque.

Si un jeton s'est perdu entre les processus P_{i-1} et P_i et si les deux jetons n'étaient pas simultanément présents en P_{i-1} lorsque l'un d'eux est parti et s'est perdu, alors cette perte et la regénération associée auront lieu en P_i, donc "au plus tôt", ce qui n'est pas le cas de l'algorithme de Misra.

Dans le cas où j et h se trouvent simultanément dans un site et le premier à le quitter se perd, il y a un moyen simple d'éviter le tour complet sur l'anneau de l'autre jeton pour détecter la perte : si h est le premier jeton à partir du site, alors lors de son départ le processus met le drapeau de j à *faux* et de cette façon ce dernier en détectant la rupture de séquence saura que h vient de se perdre (voir Section 4).

4. UN ALGORITHME A k JETONS.

Le Problème.

L'inconvénient de l'algorithme de Misra et de celui que nous venons de présenter est leur non-résistence à la perte des deux jetons. Une réponse à ce problème qui respecte le principe de l'algorithme précédent consiste à le généraliser à *k* jetons de façon à ce qu'il continue à fonctionner tant qu'il reste au moins un jeton sur l'anneau. Pour une configuration donnée, par exemple, un choix approprié de *k* pourrait rendre la probabilité de défaillance de l'algorithme aussi faible que possible. Les *k* jetons tournent tous dans le même sens dans l'anneau des processus.

Multiplier les jetons peut être également intéressant du point de vue de l'attribution des privilèges. En effet, le système distribué, qui contrôle l'ensemble des processus communicants (à l'aide d'un schéma de communication en anneau unidirectionnel) peut offrir à ces processus plusieurs ressources distinctes dont chacune est accessible en exclusion mutuelle. A chaque ressource peut alors être associé un jeton et seul le processus possesseur du jeton a le privilège de son utilisation. Ainsi l'ensemble des jetons permet une exclusion mutuelle sélective fonction de la ressource utilisée. Le système d'allocation des ressources est exempt d'interblocage dans l'hypothèse où les processus n'utilisent qu'une ressource à la fois. Dans le cas où plusieurs ressources sont simultanément utilisées par un même processus une politique d'allocation selon un ordonnancement des ressources élimine tout interblocage (schéma des classes ordonnées de Havender [HAV68]).

Principe de la généralisation.

Le but de la généralisation est d'offrir un algorithme tel que la perte d'un jeton entre les processus P_{i-1} et P_i soit détectée dès qu'un jeton actif (n'importe lequel) arrive en P_i (rapelons que les indices sur les processus sont calculés modulo $n+1$). Il est clair que, présentant des performances supérieures, le coût de ce nouvel algorithme et plus grand : ceci aparait à deux niveaux : les jetons vont véhiculer plus d'information (proportionnellement à la quantité totale de jetons en circulation) et en conséquence les calculs à effectuer dans les sites seront plus longs.

L'adaptation de l'algorithme à deux jetons doit être telle que tous les jetons aient des comportement identiques, chacun d'entre eux détectant lors

de son arivée en P_i les pertes qui se sont produites entre P_{i-1} et P_i. Pour cela, chaqu'un des processus P_i est doté de la déclaration suivante :

var *m* : **tableau** [*0*..k-1] **de entier**

$m_j[j]$ sert au jeton j à déposer sa trace lors de son passage en P_i.

Outre son identité, le jeton j véhicule maintenant deux tableaux :

v_j : **tableau** [*0*..k-1] **de entier**
s_j : **tableau** [*0*..k-1] **de boléen**

On a come précédemment :

* $v_j[j]$ est le compteur qui indique le nombre de processus visités par le jeton j depuis le début ; lorsque j arrive à P_i il éxécute :

$v_j[j] := v_j[j] + 1$;
$m_j[j] := v_j[j]$

* $v_j[h]$, pour tout $h \neq j$ sert au jeton j à mémoriser la trace laissée par le jeton h dans le processus précédent ; ces variables permetront au jeton j de détecter la perte des jetons h entre P_{i-1} et P_i.

* $s_j[h]$ est le drapeau mis à *vrai* lorsque le jeton j rencontre le jeton h dans un site et qui lui permettra, s'il part le premier, de savoir que la rupture de séquence qu'il trouve forcémment en arrivant au processus suivant sur l'anneau est due au fait qu'il a doublé h et non que celui-ci s'est perdu.

Le booléen $s_j[j]$ ne correspond à aucune information utile ; sa présence simplifie l'écriture de l'algorithme.

Il est important de remarquer que les jetons arrivent l'un après l'autre dans un site. A un instant donné, le premier jeton qui arrive en P_i détecte les pertes éventuelles entre P_{i-1} et P_i et regénère les jetons corespondants; le fait qu'il y ait un premier jeton qui arrive après une ou plusieurs pertes consécutives assure que les jetons perdus sont regénérés en exemplaire unique. Rappelons de plus que les jetons ne se doublent pas sur les lignes de communication mais peuvent s'y perdre alors que dans un processus les jetons peuvent se rencontrer et éventuellement s'y doubler.

L'algorithme.

Les variables $m_i[j]$ des processus P_i relatives à la trace que laisse le jeton j sont initialisées de la façon suivante : si le jeton j se trouve au début dans le processus P_i l'on a :

$$m_i[j] = x_j \quad \text{(valeur quelconque, } 0 \text{ par exemple)}$$
$$m_{i-1}[j] = x_j - 1$$
$$\ldots$$
$$m_{i+1}[j] = x_j - n + 1$$

Les compteurs v_j associés au jeton j (initialement situé dans P_i) sont initialisés ainsi :

$$v_j = m_i$$

Les booléens $s_j[h]$ que véhicule le jeton j sont initialisés à **vrai** ou à **faux** selon que le jeton h est également présent dans P_i ou non ; comme on l'a expliqué plus haut, $s_j[j]$ est initialisé à **faux** et gardera toujours cette valeur.

Le texte du processus P_i est le suivant ($i \in 1..n$) :

(voir page suivante)

lors de la réception de (j, v_j, s_j)

faire

$v_j[j] := v_j[j] + 1$; $m[j] := v_j[j]$; «arrivée de j»

pour tout $h \in 0..k-1, h \neq j$ **tel que** h **présent**

faire

$v_j[h] := m[h]$; $s_j[h] := vrai$; «rencontre»
$v_h[j] := m[j]$; $s_h[j] := vrai$

fait

pour tout $h \in 0..k-1$ **tel que** h **absent**

faire

si h **non perdu alors**

début

$v_j[h] := m[h]$; $s_j[h] := faux$ «mise à jour»

fin

sinon «h perdu»

début

$v_h := v_j$; $s_h := s_j$; «regénération de h par j»

$v_h[h] := v_h[h] + 1$; $m[h] := v_h[h]$; «on simule l'arrivée de h»

$v_j[h] := m[h]$; «on simule la rencontre entre j et h»

$s_j[h] := vrai$; $s_h[j] := vrai$

fin

fsi

fait

pour tout $h \in 0..k-1$ **tel que** h **a été regénéré par** j

faire

$v_h := v_j$; $s_h := s_j$; «simule la rencontre de tous les jetons regénérés entre eux»

pour tout $l \in 0..k-1, l \neq j$ **tel que** l **présent et** l **n'a pas été regénéré par** j

faire

$v_l[h] := m[h]$; $s_l[h] := vrai$ «complète la rencontre entre l et h»

fait

fait

fait

Commentaires.

Lorsque le jeton j arrive dans un processus P_i, il commence par incrémenter son compteur (nombre de processus visités) et laisse sa trace dans $m_i[j]$. Ensuite, il rencontre tous les jetons qui se trouvent dans le site lors de son arrivée : on met à jour les traces et on passe à *vrai* les booléens respectifs.

Le second bloc **pour** concerne tous les jetons qui ne sont pas dans le site. Pour chacun de ces jetons h, si h est **non perdu**, c'est-à-dire, si h ne s'est pas perdu entre P_{i-1} et P_i, j met à jour la trace correspondante et le booléen $s_j[h]$ est passé à *faux*. Si par contre h s'est perdu entre P_{i-1} et P_i, il est regénéré. Les conditions **perdu** et **non perdu** correspondent aux booléens suivants :

non perdu = ($m[h] = v_j[h] + 1$) **ou** (($m[h] \neq v_j[h] + 1$) **et** $s_j[h]$)

perdu = ($m[h] \neq v_j[h] + 1$ **et non** $s_j[h]$)

On peut noter que dans la regénération de h par j il n'est pas nécéssaire de donner toutes les valeurs à v_h et à s_h car ceci est réalisé définitivement dans le dernier bloc **pour** ; on a laissé la première ligne de ce bloc par souci de clarté.

Dans le dernier bloc on commence par copier dans les tableaux portés par chacun des jetons regénérés h (la condition **a été regénéré par** signifie évidemment "dans le bloc précédent") les informations dont j dispose à cet instant : la raison est que de cette façon, chacun des jetons h qui vient d'être regénéré rencontrera les autres jetons regénérés et aura les valeurs correctes pour leurs traces et les drapeaux ; les valeurs sont correctes parce que à la fin du bloc précédent le jeton j a rencontré tous les jetons qu'il a regénéré. Enfin, il est clair que chacun des h regénéré a "vu" les autres jetons l qui étaient présents dans le site lors de l'arrivée de j (par la mise à jour précédente) mais qu'eux n'ont pas encore connaissance de la présence des h ; le bloc **pour** imbriqué permet donc de finir cette mise à jour.

On peut mieux comprendre le principe de l'algorithme avec une version plus compacte sous la forme de la procédure recursive suivante :

```
proc réception( j, v_j, s_j )
    début

    v_j[j] := v_j[j] + 1 ; m[j] := v_j[j] :  «arrivée de j»

    pour tout h∈0..k-1,h≠j tel que h présent
        faire
        v_j[h] := m[h] ; s_j[h] := vrai ;
        v_h[j] := m[j] ; s_h[j] := vrai      «rencontre»
        fait

    pour tout h∈0..k-1 tel que h absent
        faire
        si h non perdu alors
            début
            v_j[h] := m[h] ; s_j[h] := faux   «mise à jour»
            fin
        sinon  «h perdu»
            début
            v_h := v_j ; s_h := s_j :  «regénération de h par j»
            «h est maintenant présent dans le site»
            réception( h, v_h, s_h )
            fin
        fsi
        fait
fin
```

Les paramètres (j, v_j, s_j) sont passés évidemment par adresses, et les variables sont toutes globales.

Amélioration des performances.

Supposons que les jetons j et h se trouvent ensemble dans le processus P_{i-1} et que h quitte ce processus en se perdant ; si ensuite j est le premier jeton actif en arriver au site P_i, il ne verra pas la perte de h à cause de son booléen $s_j[h]$ qui aura la valeur *vrai*. Une façon simple d'éviter cette situation est la suivante : en plus de la partie réception d'un jeton qui arrive, chaque processus éxécute l'opération suivante lorsqu'un jeton quitte le site :

lors du départ du jeton (j, v_j, s_j)
faire

pour tout $h \in 0..k\text{-}1, h \neq j$ **tel que** h **présent**
faire
$s_h[j]$:= *faux*
fait

fait

Si on reprend la situation évoquée plus haut, maintenant j arrivera en P_i avec $s_j[h]$ = *faux* et la perte de h sera détectée au plus tôt. Avec cette modification, toute perte est détectée dès qu'un jeton quelconque arrive au processus où elle s'est produite, même si le jeton arrivant se trouvait avec le jeton perdu dans le site précédent.

Borner le domaine des compteurs.

Comme dans les algorithmes précédents il est souhaitable de borner le domaine de définition des compteurs associés aux jetons. On constate que l'information utile transportée par le jeton j dans $v_j[j]$ (nombre de processus visités depuis le début) doit avoir une valeur différente dans chacun des sites visités. Il est donc possible, s'il y a n processus sur l'anneau, de borner la taille des variables $v_j[h]$ en les contraignant à appartenir à l'intervalle $0..n$; les opérations d'additions sur ces variables doivent alors être effectuées modulo $n+1$.

Variantes.

L'algorithme proposé peut facilement être modifié pour offrir des variantes mieux adaptées à certains problèmes particuliers. Selon les bessoins il est par exemple possible que seulement certains jetons détectent la perte de certains autres ; les jetons peuvent alors être classés en deux groupes : ceux qui sont associés aux ressources et ceux dont le rôle est d'assurer le contrôle de ces derniers. Les jetons peuvent aussi être structurés en anneau de telle façon que la perte du jeton j soit détectée par le jeton $j\text{-}1$; dans ce cas, il y a un gain substantiel dans la taille des jetons qui se comportent comme dans l'algorithme donné dans la Section 3.

Toutefois, la mise en oeuvre d'une variante peut être délicate, essentiellement au niveau de l'information qu'il faut donner à un jeton regénéré pour qu'il puisse à son tour remplir son rôle dans la détection des pertes. A titre d'exemple, nous montrons ici la variante de l'algorithme où les jetons sont structurés eux mêmes en anneau, comme indiqué plus haut : j détecte la perte éventuelle de $j+1$ **mod** k et le cas écheant regénère le jeton perdu.

La structure de chaque jeton est la même que celle de l'algorithme de la Section 3 : v_j est un tableau à 2 entrées indicées par j et $j+1$ **mod** k, s_j est un booléen ; par contre, comme il'y a k jetons en circulation, le tableau m de chaque site possèdera k entrées. Le texte de chaque processus est le suivant :

(page suivante)

```
lors de la réception de ( j, v_j, s_j )

      faire

            «arrivée de j»
      v_j[j] := v_j[j] + 1 ; m[j] := v_j[j] ;   «arrivée de j»

      h := j+1 mod k ;
      si h présent alors
            début
            v_j[h] := m[h] ; s_j := vrai     «rencontre»
            fin
      sinonsi h non perdu alors
                  début
                  v_j[h] := m[h] ; s_j := faux   «mise à jour»
                  fin
            sinon   «h=j+1 perdu»
                  début
                        «regénération de h=j+1 et simulation de
                            son arrivée»
                  v_j+1[j+1] := v_j[j+1] + 1 ;
                  m[j+1] := v_j+1[j+1] ;
α                 v_j+1[j+2] := m[j+2] ;
                  si j+2 présent alors
                        début  s_j+1 := vrai fin
                  sinon
                        début  s_j+1 := faux fin
                  fsi ;
                        «rencontre : j "voit" j+1»
                  v_j[j+1] := m[j+1] ; s_j := vrai

                  fin
            fsi
      fsi

      fait
```

Le seul point différent par rapport aux algorithmes déjà présentés se trouve à la ligne α. Lors de la regénération de $j+1$ il faut trouver une valeur correcte pour la variable $v_{j+1}[j+2]$, sachant que le jeton j ne transporte pas la trace laissée par $j+2$ dans le site précédent. Si le jeton $j+2$ se trouve dans le site où la regénération de $j+1$ a lieu, alors on aura à cet instant $m[j+2] = v_{j+2}[j+2]$ et donc à la ligne α $v_{j+1}[j+2]$ reçoit la valeur actuelle de $v_{j+2}[j+2]$; si par contre $j+2$ est absent, $j+1$ possèdera la valeur de la trace que le premier a laissée lors de son dernier passage dans le site.

On peut remarquer que si les jetons $j+1$ et $j+2$ se sont perdus entre les sites P_{i-1} et P_i, lorsque j arrive en P_i il regénère $j+1$ mais que celui-ci ne pourra faire de même avec $j+2$ qu'au boût d'un tour sur l'anneau. Il est possible d'avoir des regénérations "en cascade", mais alors il faut que j transporte l'information nécessaire pour que $j+1$ une fois regénéré soit capable de voir la perte de $j+2$, et ainsi de suite. Pour cela il faut que les jetons transportent le même tableau v à k entrées déclaré pour l'algorithme général présenté plus haut.

Cas particulier : les jetons ne se doublent pas.

Il est intéressant de considérer le cas très particulier où les jetons qui circulent sur l'anneau des processus ne se doublent pas. Dans un souci de simplification d'écriture on suppose que les jetons circulent sur l'anneau dans l'ordre de leurs identités : $0, 1, \ldots, k-1$. Dans ce cas, la seule information qui doit véhiculer un jeton est son identité et l'algorithme de détection est trivial: le tableau m_i du processus P_i se résume alors à une seule variable et il doit receptionner successivement les valeurs $0, 1, \ldots, k-1, 0, 1, 2, \ldots$ etc. ; un trou dans la numérotation permet de détecter la perte d'un ou de plusieurs jetons et de les regénérer en conséquence. Les variables m_i des processus sont alors bornées par 0 et $k-1$, k étant le nombre de jetons.

5. CONCLUSIONS.

L'algorithme de détection de la perte de jetons et de leur regénération proposé (à 2 ou à *k* jetons) est intéréssant par le fait qu'il n'utilise ni horloge de garde, ni identité des processus. Comme celui de Misra, il se démarque en cela des autres solutions. Le principe sur lequel il est fondé est toutefois différent de celui de l'algorithme de Misra : on utilise des compteurs croissants et les traces qu'ils laissent dans les processus lors de leurs passages. Cela permet de considérer les traces de chacun des compteurs comme des séquences croissantes qui doivent être sans trou : une perte a alors commme conséquence une rupture dans la séquemce correspondant au jeton perdu, rupture qui est utilisée pour la détection de cette perte. De plus, les propriétés des compteurs permettent de les faire évoluer modulo $n+1$ (n étant le nombre de processus) ce qui en autorise une mise en oeuvre simple. La version à 2 jetons a une performance supérieure à celle de l'algorithme de Misra puisque dans ce dernier une perte n'est pas détectée dans le processus qui aurait dû récevoir le jeton perdu mais a priori dans un site quelconque dans l'anneau, dépendant de l'état du système lorsque la perte s'est produite. Dans notre algorithme, dès que le jeton actif arrive au site en question, il détecte cette perte avec le gain évident de temps et de sécurité, gain qui sera d'autant plus important que le nombre de processus sera plus grand. Le principe qui consiste à utiliser des séquences croissantes qui doivent être sans trou a également été utilisé pour produire un algorithme distribué de prévention de la dérive mutuelle entre n horloges logiques ([RAY 85]).

REFERENCES :

CHA 79 CHANG E.J., ROBERTS R.
An Improved Algorithm for Decentralized Extrema Finding in Circular Configuration of Processors.
Comm. ACM, Vol.22,5,(May 1979),pp.281-283.

DOL 82 DOLEV D., KLAWE M., RODEV M.
An O(nlogn) Unidirectional Distributed Algorithm for Extrema Finding in a Circle.
Journal of Algorithms, Vol.3,(1982),pp.245-260.

HAV 68 HAVENDER J.W.
Avoiding Deadlocks in Multitasking Systems.
IBM System Journal, Vol.7,2,(1968),pp.36-38.

LEL 77 LE LANN G.
Distributed Systems : Towards a Formal Approach.
IFIP Congress, Toronto, August 1977, pp.155-160.

LEL 78 LE LANN G.
Algorithms for Distributed Data-Sharing Systems Which Use Tickets.
Proc. 3rd Berkeley Workshop on Distributed Data Base and Computer Networks,(August 1978),pp.259-272.

MIS 83 MISRA J.
Detecting Termination of Distributed Computation Using Markers.
Proc. of the 2d annual ACM Symposium on Principles of Distributed Computing, Montreal,(August 1983),pp.290-294.

RAY 85 RAYNAL M.
A Distributed Algorithm to Prevent Mutual Drift Between n Logical Clocks.
Proc. of Int. Workshop on Distributed Systems, Newcastle,(April 1985).

RIC 83 RICART G., AGRAWALA A.K.
Author's response to "On Mutual Exclusion in Computer Networks" by Carvalho and Roucairol.
Comm. ACM,Vol.26,2,(February 1983),pp.147-148.

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