2.3.2 Algorithme de Ricart-Agrawala (1981)

L'idée de cet algorithme est assez proche de l'algorithme précédent mais on va essayer de réduire le nombre de messages nécessaires à l'accès de la section critique : lorsqu'un processus P_i communique à P_j son souhait d'accéder à la SC, P_j ne lui retourne un accusé de réception `ok` que si P_j ne souhaite pas y accéder ou si P_j n'est pas prioritaire (voir ci dessous). Dans le cas où P_j ne répond pas tout de suite, il le fera après avoir accédé à la SC (il utilise un tableau `Waiting` pour stocker la liste des processus à qui il doit envoyer un `ok` plus tard).

La priorité d'un processus est établie par un numéro que chaque processus choisit lorsqu'il souhaite accéder à sa SC. **Plus ce numéro est petit, plus sa demande d'accès à la SC est prioritaire.** Ce numéro est choisi supérieur à tous les autres numéros que le processus a vu pour le moment (on utilise pour cela une variable `maxnb`).

Un processus n'accède alors à sa SC que lorsqu'il a reçu un accord explicite de tous les autres processus ($N-1$ messages).

Cet algorithme utilise deux types de message : `request` (comme précédemment) et `ok` qui sert à donner son accord à un processus pour qu'il accède à la SC. Notons enfin qu'ici nous ne faisons pas l'hypothèse que les messages arrivent dans l'ordre : si P_i envoi un premier message à P_j, puis un second, alors l'ordre d'arrivée n'est pas fixé.

```
Algorithme de Ricart-Agrawala
boolean Waiting[1..n] := False
boolean reqCS :=False
int pr := 0
int maxpr := 0      // plus grand numero reçu
int nba:=0          // compte le nb de réponses attendues

-- Processus Pi
loop forever :
p1: section NC
p2: { reqCS := True
P3:   nba := N-1
p4:   pr := maxpr +1 }
P5: for all j != i : Send(request,pr,i) -> j
p6: await [ nba == 0 ]
p7: section critique
p8: { reqCS := False
p9:   for all j: if (Waiting[j] == True) :
p10:          Waiting[j] := False
p11:          Send(ok,i) -> j }
```

FIG. 13 – Algorithme de Ricart-Agrawala

Dans le code des figures 13 et 14, on utilise des « {...} » pour désigner des instructions que l'on fait de manière atomique : le code de la partie principale et celui de la gestion des réceptions ne peuvent pas alors s'entremêler (pour garantir cette exclution mutuelle on peut utiliser les algorithmes vus précédemment basé sur le partage de la mémoire comme Peterson). On verra pourquoi cette hypothèse est nécessaire.

```
case (request,k,j)  :
  { if (k > maxpr) :  maxpr := k
    if (!reqCS or (k,j)<<(pr,i)) : Send(ok,i) -> j
      else : Waiting[j] := True }

case (ok,j)  :
  nba := nba - 1
```

FIG. 14 – Algorithme de Ricart-Agrawala – gestion des réceptions

Théorème 8 *L'algorithme de Ricart-Agrawala assure l'exclusion mutuelle de la section critique : la propriété* $\Box \bigwedge_{i \neq j} \neg(p\mathcal{7}^{(i)} \wedge p\mathcal{7}^{(j)})$ *est vérifiée.*

Preuve : Supposons que P_i et P_j se trouvent ensemble dans leur section critique. Il se sont donc chacun envoyés un message `ok`. Ils ont aussi chacun choisit une priorité : pr_i pour P_i et pr_j pour P_j.

Pour P_i on distingue les dates suivantes :
- t_i^0 où P_i choisit son numéro `pr` ;
- t_i^1 où P_i envoie le message `(request,i,pr)` à P_j ;
- t'_j où P_j reçoit `(request,i,pr)` et retourne `(ok,j)` à P_i (NB : en raison des hypothèses d'atomicité, on sait que cet instant existe : la réception du message de P_i est suivie immédiatement – sans l'exécution d'instructions de la procédure principale de P_j – par l'envoi de `ok` à P_i) ;
- t_i^2 où P_i reçoit le message `(ok,j)` de P_j.

On définit de même t_j^0, t_j^1, t'_i et t_j^2 pour le processus P_j. On a clairement : $t_i^0 < t_i^1 < t'_j < t_i^2$, et $t_j^0 < t_j^1 < t'_i < t_j^2$.

Supposons $t_i^0 < t_j^0$. Alors on distingue deux cas :
- $t'_j < t_j^0$: alors le choix de pr_j sera fait après la mise à jour de `maxpr` dans P_j et donc on aura $(pr_j, j) >> (pr_i, i)$. En conséquence, il n'est pas possible que P_i envoie un message `ok` à P_j avant de quitter sa SC... Il ne peut y avoir de conflit en section critique de cette manière.
- $t_j^0 < t'_j$: alors on a soit $(pr_j, j) >> (pr_i, i)$ ou $(pr_i,) >> (pr_j, j)$. Dans le premier cas, P_j ne pourra recevoir de message `ok`, et dans le second c'est P_i qui n'en recevra pas de la part de P_j. Dans les deux cas, un des deux processus devra attendre que l'autre sorte de la section critique.

Le cas $t_j^0 < t_i^0$ est similaire.

Il n'y a donc pas de conflit en section critique. □

L'absence de famine est aussi garantie :

Théorème 9 *L'algorithme de Ricart-Agrawala garantit l'absence de famine sous hypothèses...*
- *d'équité entre processus,*
- *de terminaison des sections critiques,*
- *de gestion ininterrompue des réceptions de messages, et*
- *de respect des blocs atomiques mentionnés dans l'algorithme.*

Alors l'algorithme vérifie : $\Box \bigwedge_i (p\mathcal{2}^{(i)} \Rightarrow \Diamond\, p\mathcal{7}^{(i)})$.

Preuve : Supposons que le processus P_i soit en situation de famine. Soit pr_i sa priorité. Si P_i reste bloqué, c'est forcément en `p6` : il attend d'avoir une réponse de chaque processus [1].

Il existe donc un processus P_j qui ne donne jamais son accord à P_i. Si tel est le cas, c'est que P_j veut aussi accéder à sa SC (ceci est nécessaire pour différer l'envoi de `ok`). Et si P_j bloque pour toujours P_i, c'est que lui aussi est bloqué : car si P_j accédait à sa SC, il la terminerait et finirait par envoyer `ok` à P_i...

Notons au passage, que les autres processus finiront par être aussi bloqués lors de leur prochain pré-protocole : ils ne recevront plus de message `ok` de P_i et P_j car ils auront des numéros supérieurs à pr_i et pr_j... Tout le monde finira donc bloqué (ou en section non-critique).

Soit P_m le processus bloqué dans son pré-protocole ayant la plus petite priorité pr_m (*i.e.* minimale pour $<<$) . Soit P_k un autre processus bloqué qui n'a pas envoyé son message `ok` à P_m. Pourquoi P_k a-t-il bloqué P_m ? C'est qu'au moment de la réception de `(request,`pr_m`,m)`, P_k voulait accéder à la SC (`reqCS==True`) et que son pr était inférieur (pour $<<$) à pr_m (NB : c'est ici que l'hypothèse d'atomicité du bloc `p2`,`p3`,`p4` est utilisée). Mais depuis P_k a changé de pr (puisque pr_m est le minimum – l'égalité est impossible grâce à la relation $<<$), c'est donc qu'il est passé en SC et donc qu'il a exécuté la boucle `p9` et donc envoyé un message `ok` à P_m. Il y a bien contradiction : P_m ne peut pas être bloqué... □

Dans cet algorithme, une demande d'accès en SC demande $2(n-1)$ messages.

Exercice 4 *Donner une exécution du protocole où le non respect des hypothèses d'atomicité conduit à un blocage et une famine.*

On peut trouver une preuve automatique de cet algorithme dans [3] (accessible sur le web).

[1] son compteur `nba` décroît à chaque réception d'un `ok` et il est aisé de voir que l'envoi d'un message `ok` ne se fait qu'à la réception d'un message `request` dans le bloc de gestion des messages, ou après la section critique **sinon** : un processus ne peut pas envoyer plus de `ok`.