
Correction des TD d'AlgoD

Année 2007

Version 2.1

Université de Montpellier
Place Eugène Bataillon
34095 Montpellier Cedex 5

RODOLPHE GIROUDEAU
161, RUE ADA
34392 MONTPELLIER CEDEX 5
TEL : 04-67-41-85-40
MAIL : RGIROU@LIRMM.FR

Algorithme distribué
TD – Séance n° 0

Exercice 1 – Horloges de Lamport et de Martell

Nous considérons les horloges de Lamport définies de la manière suivante :

Une horloge est une fonction θ définie à partir des événements vers un ensemble ordonné tel que $a \prec b \Rightarrow \theta(a) < \theta(b)$. L'horloge de Lamport θ_L affecte à chaque événement a la longueur k de la plus longue chaîne de causalité $b_1 \prec \dots \prec b_k = a$.

Un algorithme distribué permet de calculer θ_L et sera basé sur les caractéristiques suivantes :

- Si a est un événement interne ou l'envoi de messages, soit k la valeur de l'horloge de l'événement précédent ($k = 0$ si il n'y a pas de événement précédent) $\theta_L(a) = k + 1$;
- Si a est l'événement de réception, k la valeur de l'horloge de l'événement précédent ($k = 0$ si il n'y a pas de événement précédent), et b la valeur de l'horloge de l'événement a , alors $\theta_L(a) = \max\{k, \theta_L(b)\} + 1$.

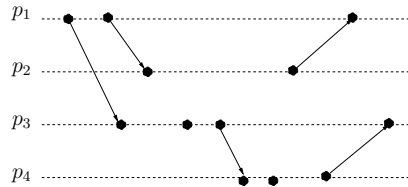


FIG. 1 – Sur les horloges de Lamport

1. Donner l'algorithme qui permet de calculer les horloges logiques.
2. Donner les valeurs des horloges pour la figure 1. Avons-nous un ordre total ?
3. Que devons rajouter à la définition précédente pour créer un ordre total ?
4. Montrer que si $e \rightsquigarrow e'$ alors $H(e) \prec H(e')$. Utiliser par récurrence sur la longueur n de la suite d'événement reliant e à e'
5. Considérons maintenant la notion de vectorielle proposé par Mattern.

- Chaque site i gère un vecteur d'entiers de n éléments (une horloge HV_i avec n le nombre de sites ;
- Chaque message m envoyé est estampillé (daté) (EV_m) par la date de son évènement :
 - Si un événement est locale, alors $HV_i[i] \leftarrow HL_i[i] + 1$;
 - Si l'évènement est l'envoi d'un message m alors

$$\begin{cases} HV_i[i] \leftarrow HL_i[i] + 1 \\ EV_m \leftarrow HV_i \end{cases}$$

- Si l'évènement est la réception du message m alors

$$\begin{cases} HV_i[i] \leftarrow HL_i[i] + 1 \\ HL_i[i] \leftarrow \max(HL_i[j], EL_m[j]), j \neq i \end{cases}$$

- (a) Donner l'évolution des horloges vectorielles pour le graphe de la figure 2.

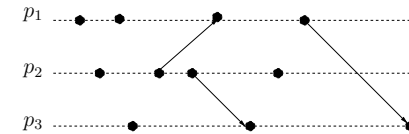


FIG. 2 – Sur les horloges vectorielles

- (b) Montrer que la valeur de la i ème composante de EV_e correspond au nombre d'évènements du site S_i appartenant au passé de e

$$\forall i, EV_e[i] = \text{card}(\{e' : e' \in S_i \wedge e' \rightsquigarrow e\})$$

- (c) Compléter la relation d'ordre suivante :

$$EV_e \preceq EV_{e'} \text{ ssi}$$

- (d) Montrer que $e \rightsquigarrow e'$ ssi $EV_e \preceq EV_{e'}$.
- (e) Montrer que $e \parallel e'$ ssi $EV_e \parallel EV_{e'}$ où $e \parallel e'$ désigne deux évènements e et e' sont concurrents.
- (f) Cet ordre permet-il la délivrance causale ?

1. L'algorithme est le suivant :

En cas d'un événement interne

$$\theta_p := \theta_p + 1;$$

En cas d'un événement d'envoi de message

$$\theta_p := \theta_p + 1;$$

Envoyer($\langle \text{message}, \theta_p \rangle$);

En cas d'un événement de réception de message ($\langle \text{message}, \theta \rangle$)

$$\theta_p := \max\{\theta_p, \theta\} + 1;$$

2. La solution est donnée par la figure 3. L'ordre des événements n'est pas un ordre strict : plusieurs événements peuvent porter la même valeur.

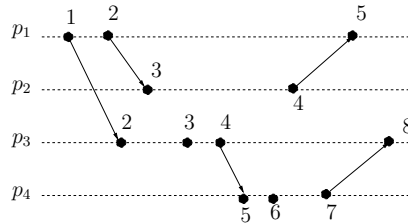


FIG. 3 – Solution

3. l'horloge logique du site qui envoie le message. On a la relation d'ordre suivante $HL(e)$ de e sur le site i est un couple (h_i, i) :

$$(h_i, i) < (h_j, j) \Leftrightarrow (h_i < h_j \text{ ou } (h_i = h_j \text{ et } i < j))$$

4. Par récurrence :

– si $n = 0$ alors $e = e'$ et $HL(e) \preceq HL(e')$

– Si $n > 1$ soit $e = e_0, e_1, \dots, e_n = e'$ une suite d'évènement tel que $\forall i \in [0, \dots, n-1], e_i \rightarrow e_{i+1}$.

Par récurrence on a $e \rightsquigarrow e_{n-1}$ et $HL(e) \prec HL(e_{n-1})$. Sachant que $e_{n-1} \rightarrow e_n$, on a

– soit e_{n-1}, e_n appartiennent au même site $HL(e_{n-1}) \prec HL(e_n)$;

– Soit il existe une message m tel que $send(m) = e_{n-1}$; par définition on a $HL(e_{n-1}) \prec HL(e_n)$

– Par transitivité de \preceq on a $HL(e) \prec HL(e_n)$

5. Sur l'approche vectorielle

(a) La solution est donnée par la figure 4.

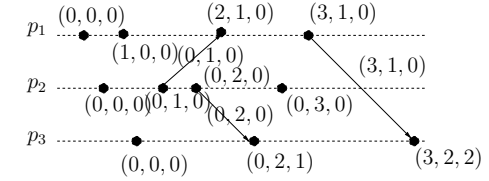


FIG. 4 – Solution pour les horloges vectorielles pour la figure 2

(b) Soit $e' \in PASSE(e)$ un événement du site $S_i : EV_{e'}[i] \leq EV_e[i]$ où $PASSE(e)$ désigne un ensemble des événements antérieurs à e dans l'ordre causal (e appartient à cet ensemble).

Considérons e' tel que $EV_{e'}[i] = EV_e[i] \cdot \text{card}(PASSE(e') \cap \{e'' \text{ appartenant à } S_i\}) = EV_e[i] - 1$. $FUTUR(e') \cap PASSE(e) = \emptyset$ sinon $EV_{e'}[i] \neq EV_e[i]$ où $FUTUR(e)$: ensemble des événements postérieurs à e dans l'ordre causal (e appartient à cet ensemble).

(c)

$$EV_e \preceq EV_{e'} \text{ ssi } \forall i, EV_e[i] \leq EV_{e'}[i]$$

(d) Si $e \rightsquigarrow e'$ alors

- $PASSE(e) \subseteq PASSE(e')$
- $\forall i, EV_e \leq EV_{e'}$
- $EV_e \subseteq EV_{e'}$

(e) $e \parallel e'$ alors : soit S_i et S_j les sites appartenant à e et e'

- si $j = i$ alors, e et e' ne peuvent pas être concurrents
- donc $j \neq i$
- Soit $x(e, j)$ le plus grand élément du $PASSE_j(e)$
- on a $x(e, j) \rightarrow e'$ (car si $e' \rightsquigarrow x(e, j)$, alors $e' \in PASSE(e)$)
- donc $PASSE_j(e) = PASSE_j(x(e, j)) \subset PASSE_j(e')$
- Par conséquent : $EV_e[i] < EV_{e'}[i]$

(f) Non il suffit de considérer le graphe donné par la figure ??

Fin correction exercice 1

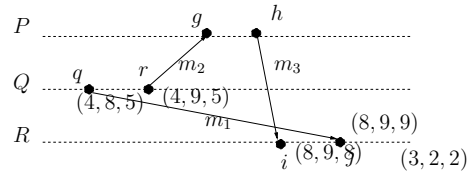


FIG. 5 – Contre-exemple

Exercice 2 – La diffusion

Nous considérons le problème de la diffusion d'un sommet r_0 vers tous les autres.

Si $i = r_0$ alors Envoyer($\langle message, v \rangle$) avec $v \in voisin_{r_0}$;

Lors de la réception ($\langle message, v \rangle$) de i ;

Envoyer($\langle message, v \rangle$) avec $v \in voisin_j \setminus \{i\}$;

1. Quel est l'inconvénient de l'algorithme précédent ?
2. Proposer une solution pour éviter la non terminaison.
3. Donner une borne inférieure en fonction de n pour tout algorithme de diffusion.
4. De même en fonction de m .
5. Donner la borne supérieure de votre algorithme.

Correction exercice 2

1. L'inconvénient porte sur la non terminaison garantie
2. Voici le nouvel algorithme :
Si $i = r_0$ alors Envoyer($\langle message, v \rangle$) avec $v \in voisin_{r_0}$;

Lors de la réception ($\langle message, v \rangle$) de i ;

Si $First := true$ alors Envoyer($\langle message, v \rangle$) avec $v \in voisin_j \setminus \{i\}$; $First := false$;

3. Il faut que les $n - 1$ autres sommets soient avertis.

4. Il faut passer par au moins toute les arêtes. Supposons le contraire, il existe donc une arête $[x, y]$ non traversé par le message M . Soit G_1 (resp. G_2) le graphe issue de G dans lequel nous avons rajouté un sommet z_1 (resp. z_2) entre x_1 et y_1 sur l'arête $[x_1, y_1]$ (resp. x_2 ; Y_2 et $[x_2, y_2]$). On relie les deux copies par z_1 et z_2 . En appliquant l'algorithme sur G' une des copies ne sera pas inondée, contradiction avec le principe que tous les sommets sont touchés.

Plus simple rajouter un sommet z entre x et y , et sachant que l'arête $[x, y]$ n'est pas parcouru alors le sommet z ne peut être atteint.

5. La borne supérieure est $2m - (n - 1)$. En effet, les arêtes sont traversés deux fois sauf celles dont le père envoie. Sachant que nous avons qu'un seul père et que le plus petit graphe connexe est un arbre donc le résultat s'impose.

Fin correction exercice 2

Exercice 3 – Asservissement mutuel d'horloges logiques pour deux sites

Le problème est le suivant : on veut contraindre les deux processus de telle façon qu'à tout instant chacun d'eux ne puisse pas émettre plus de δ requêtes d'avance par rapport à l'autre. Vue l'association des valeurs d'horloges aux requêtes, le problème consiste à maintenir invariante la relation suivante : $|h_i - h_j| \leq \delta$.

Cette relation est vraie : $h_i = h_j = 0$.

Les messages relatifs à ces échanges vont être véhiculés par des lignes que nous supposons dotées du comportement suivant : les messages peuvent être dupliqués, déséquilibrés et perdus.

Pour cela on introduit à tout instant dans chaque processus P_i une variable locale max_i qui lui indique à tout instant la valeur jusqu'à laquelle il peut faire croître son horloge h_i sans que cela n'invalidé l'invariant ; la condition associée à l'appel à dater est donc $h_i < max_i$.

Conditions initiales

–

Lors de l'appel à dater

Si $h_i < max_i$

$h_i := h_i + 1$

Envoyer($\langle h_i \rangle$) à P_j

1. Donner la procédure de réception.
2. Montrer que nous avons bien $|h_i - h_j| \leq \delta$ tout le long du processus.
3. Nous supposons que les messages peuvent être perdus et dupliqués mais ne peuvent

être déséquilibrés. Simplifier le code de la réception d'un message.

4. Comment peut-on limiter la croissance des horloges? Soit x la valeur envoyée du site h_i à un instant donné. Donner un encadrement de x par rapport δ et h_i . Donner maintenant l'algorithme qui prend en compte le non dérive des horloges.

Correction exercice 3

1. La procédure de réception est la suivante :

Lors de la réception de (x) de P_j

$\max_i := \max(\max_i, x + \delta)$

2. Il est facile de prouver cet algorithme de contrôle. Le prédicat $|h_1 - h_2| \leq \delta$ est vrai initialement ; il s'agit de montrer, par induction que toutes les opérations associées aux événements qui peuvent survenir le laissent vrai. Par construction la séquence de valeurs émises par P_i et considérées par P_j dans la mise à jour de \max_j est croissante, soient x_i et x_j les dernières de ces valeurs. Nous avons :

$x_i \leq h_i \leq \max_i$ et $x_j \leq h_j \leq \max_j$

On en déduit que $h_i - h_j \leq \max_i - x_j$ et $h_j - h_i \leq \max_j - x_i$

Or les mises à jour de \max_i sont elles que :

$\max_i = x_j + \delta$ et donc le prédicat est vérifiée

3. La procédure de réception simplifiée :

Lors de la réception de (x) de P_j

$\max_i := x + \delta$

4. Pour éviter la dérive des horloges nous pouvons introduire un modulo (modulo $2\delta + 1$).

On a $x - \delta \leq h_i \leq x + \delta$. En effet, on a $h_i - x \leq \delta$ et $h_i \geq x - \delta$ car $h_j \geq x$ et $h_j - h_i \leq \delta$. Ce qui peut être reformulée : lorsque P_i reçoit une valeur x celle-ci s'écarte d'au plus de δ de h_i et ne peut donc être que l'une des $2\delta + 1$ valeurs différentes d'un intervalle défini par h_i et δ : $h_i - \delta \leq x \leq h_i + \delta$.

Le domaine de définition d'une valeur x de message étant parfaitement défini en fonction de δ et h_i (puisque'il n'y a pas de déséquilibrage) lorsque ce message est reçu par P_i , il est possible de ne communiquer que des valeurs modulo k avec $k \geq 2\delta + 1$. Lorsque P_i reçoit une telle valeur x il doit alors démoduler pour mettre à jour sa variable \max_i en calculant le nombre m tel que $h_i - \delta \leq km + x \leq h_i + \delta$.

Contraintes initiales constant $k = 2\delta + 1$, var $m : 0 \dots \infty$

Lors de l'appel à dater

Si $h_i < \max_i$

$h_i := h_i + 1$

Envoyer $(\langle h_i \text{ mod } k \rangle)$ à P_j

Lors de la réception de (x) de P_j

Soit m la valeur telle que $h_i - \delta \leq km + x \leq h_i + \delta$

$\max_i := km + x + \delta$

il suffit donc $\log_2(2\delta + 1)$ bits suffisent pour représenter n'importe quelle valeur véhiculée par les messages de contrôle. Les lignes sont moins chargées. Le transfert se fait sur les sites.

Fin correction exercice 3

Algorithme distribué
TD – Séance n° 1

Cette série d'exercices est consacrée au problème de l'élection. A un moment donné, lors d'une exécution ou après une panne (réparée) les sites ont tous une valeur (deux à deux distinctes). Le but de l'élection est de déterminer le site qui a la plus grande valeur. Ce site est le *gagnant*. A la fin de l'élection, chaque site doit connaître ce gagnant. Nous supposons dans un premier temps que le réseau est un cycle ou un circuit. Nous supposons que les *identificateurs* (adresse) des sites transitent avec les messages (champ source du message) et que les liens sont FIFO. Au début de l'algorithme, chaque site i a une valeur (locale) val_i qui va servir de base à l'élection (sans perte de généralité on peut confondre cette valeur avec l'identificateur du site). Chaque valeur est unique. Le but d'un algorithme d'élection est de faire en sorte qu'au bout d'un temps fini, tous les sites soient d'accord sur le site élu.

Exercice 1 – Anneau La solution la plus simple au problème de l'élection dans un cycle unidirectionnel est décrite de manière informelle et incomplète par :

- Au début, chaque site i maintient une variable max_i initialisée à val_i puis il envoie sa valeur val_i vers son voisin.
 - Chaque fois qu'un site i reçoit une valeur V il fait $max_i := \max\{max_i, V\}$ et fait suivre V vers son autre voisin.
1. Si un seul (ou plusieurs) site initie l'élection, comment tous les autres sites peuvent participer à l'élection ? (problème du réveil).
 2. Quelle est la condition à rajouter pour que l'algorithme se termine ? (problème de la terminaison). Qui sait qui a gagné l'élection, comment proclamer le résultat ? (problème de la prise de décision).
 3. Combien de messages transitent durant cet algorithme ?
 4. Ecrire l'algorithme.
 5. Donner une exécution sur le graphe donné par la figure 1.
 6. Est-ce que cet algorithme reste valide en mode asynchrone ?
 7. Que se passe-t-il si on envoie le max ?

Correction exercice 1

1. Pour le réveil des sites on combine un réveil spontané et un réveil qui a lieu lors de la réception d'un message lié au protocole de l'élection. Dans ce dernier cas, les sites initialisent max_i et font tourner l'algorithme.
2. On suppose que les valeurs sont deux à deux différentes. Lorsqu'un site i reçoit sa propre valeur de son voisin, c'est le signe qu'il a reçu toutes les valeurs. Il peut alors arrêter de retransmettre.

Le site i connaît alors le site ayant la valeur maximale (gardée dans max_i). Tous les autres sites vont finir par être aussi dans cet état. Il est alors inutile de proclamer le résultat.

3. Supposons que nous ayons n sites. Chaque site envoie son message qui fait le tour du cycle et parcourt donc n arcs. Cela fait donc n^2 messages échangés en tout.
4. Non l'algorithme ne fonctionne pas en mode asynchrone. Pour qu'il marche il faut mettre un compteur sur chaque site qui correspond au nombre de site dans l'anneau. Et tant que je n'ai pas reçu n messages et le processus n'est pas terminé.
5. Voici l'algorithme :

```
En cas de réveil spontané
  participanti := Vrai;
  maxi := vali
  Envoyer(< ELECTION, vali >);
```

```
Lors de la réception d'un message < ELECTION, V >
  maxi := max{maxi, V}
  Si vali = V alors
  Arrêt(succès)          sinon Envoyer(< ELECTION, V >);
```

6. Si on envoie le max l'algo n'est pas trivial.

Fin correction exercice 1

Exercice 2 – Anneau, Algorithme de Chang-Roberts

Nous proposons maintenant un autre algorithme pour l'élection dans un cycle unidirectionnel. Chaque site i maintient un booléen $participant_i$ qui est initialisé à *Faux*. Le code pour le site i est le suivant.

```
En cas de réveil spontané
  participanti := Vrai;
  Envoyer(< ELECTION, vali >);
```

```
Lors de la réception d'un message < ELECTION, V >
  Si (V > vali) alors
    participanti := Vrai;
    Envoyer(< ELECTION, V >);
  Si (V < vali) et (non participanti) alors
    participanti := Vrai;
    Envoyer(< ELECTION, vali >);
  Si (V = vali) alors Envoyer(< ELU, vali >);
```

```
Lors de la réception d'un message < ELU, V >
  Le gagnant est le site V;
  participanti := Faux;
```

si $(val_i \neq V)$ alors Envoyer($\langle ELU, V \rangle$);

- Proposez une exécution dans la configuration de la figure 1. Qui détecte le gagnant et comment est-il propagé?
- Est-ce que cet algorithme fonctionne en mode asynchrone?
- Supposons que tous les sites se réveillent simultanément. Évaluez la complexité en nombre de messages échangés dans les configurations particulières suivantes en mode asynchrone et synchrone :
 - Les valeurs sont ordonnées croissantes dans le sens du cycle.
 - Les valeurs sont ordonnées décroissantes dans le sens du cycle.
- Nous allons étudier la complexité en moyenne.
 - Montrer qu'un site actif possédant la j ème plus grande identité parcourt en moyenne n/j liens.
 - Avec x initiateurs, donner la formule.
 - Conclure que la complexité est en $O(n \log n)$.

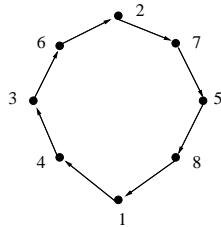


FIG. 1 – Une configuration dans C_8 .

Correction exercice 2

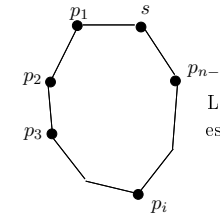
- Ici le réveil est fait soit spontanément soit par la réception d'un message du protocole. Pendant l'algorithme, un site recevant une valeur V ne va la faire suivre que si elle est supérieure à la sienne propre. Il y a donc un mécanisme de **filtrage** des valeurs. Ce qui réduit la complexité en nombre de messages par rapport à l'exercice. Qui détecte le gagnant? C'est le gagnant lui-même!! S'il reçoit sa propre valeur c'est que celle-ci n'a jamais été arrêtée (filtrée); c'est donc la plus grande. Il informe donc les autres grâce au deuxième message.
- Oui ici cet algorithme fonctionne en mode asynchrone.
- En mode synchrone :** Dans le premier cas, les messages seront stoppés très rapidement; $n - 1$ valeurs seront stoppées après une seule communication (car elles vont sur un site qui a une valeur plus grande). Seule la plus grosse valeur fait tout le tour du cycle. Ainsi, sans

compter la proclamation du résultat on a $n - 1 + n = 2n - 1$ messages échangés.

Dans le second cas, le message contenant la valeur i ème valeur sera renvoyée i fois. Ainsi le nombre total de messages échangés sera de $\sum_{i=1}^{i=n} i = \frac{n(n+1)}{2}$. **En mode asynchrone :** $2n$ si le plus grand est le seul à se réveiller (cas croissant) et $3n - 1$ dans le cas décroissant si le site $n - 1$ se réveille. Notons que nous comptons la proclamation de l'élection.

- Étudions maintenant la complexité en moyenne

- Ce résultat est obtenu simplement et en toute généralité en considérant $0 < a \leq n$ processus initiateurs déclenchant indépendamment mais pas (nécessairement) simultanément l'exécution de l'algorithme. Puisque les permutations des identités des processus sont ordonnées de manière équiprobable sur l'anneau, le jeton envoyé par un processus actif possédant la j ème plus grande identité parcourt en moyenne n/j liens sur l'anneau avant d'être éteint par un processus d'identité supérieure.
- Par conséquent, avec a initiateurs indépendants, l'élection est réalisée en utilisant un nombre moyen de jetons exactement égale à $\sum_{j=1}^a n/j = nH_a$.
- et si tous les processus sont initiateurs ($a = n$), on retrouve le résultat précédent : $O(nH_n) = O(n \log n)$.



Le sens de circulation des messages est le sens des aiguilles d'une montre

FIG. 2 – Arrangements des identités sur un anneau.

Fin correction exercice 2

Exercice 3 – Anneau bi-directionnel, Algorithme de Hirschberg et Sinclair (80) Nous allons maintenant travailler sur des cycles bi-directionnels (on peut envoyer des messages dans les deux sens sur le cycle). L'algorithme proposé marche par *phase* et est basé sur le filtrage. Les sites peuvent être dans deux états *actif* ou *perdant*. Lorsqu'un site reçoit un message contenant une valeur plus grande que la sienne il devient *perdant*. Sinon il reste *actif*.

A la phase i , ($i = 0, 1, \dots$) les sites encore actifs envoient leur valeur à une distance de 2^i d'eux. Lorsqu'un site *actif* reçoit un tel message il devient *perdant* si la valeur reçue est supérieure à la sienne sinon il passe à la phase suivante.

Nous allons supposer ici que le réseau est *synchrone par phase*. C'est-à-dire que les sites encore actifs au début de la phase i commencent tous *en même temps* à exécuter les opérations de cette phase.

- Donner un exemple d'exécution sur la graphes de la figure 1

- Après la phase i , quelle est la distance minimale entre deux sites *actifs* ?
- Combien y a-t-il de phases ? En déduire le nombre de messages échangés pendant l'algorithme.
- Qui connaît le gagnant ?
- Donner l'algorithme.
- Quel est l'inconvénient sur le nombre de messages ?

Correction exercice 3

- Au début de la phase i , la distance entre deux sites actifs est au moins de $2^{i-1} + 1$. En effet, un site reste actif après la phase $i - 1$ s'il a pu éliminer tous les sites à distance 2^{i-1} . On remarquera qu'au début de la phase i il existe au plus un site *actif* à distance 2^i à droite (et à gauche) d'un site actif (celui-ci est à distance au moins $2^{i-1} + 1$ et au plus 2^i). Ainsi, au début de la phase i , il reste au maximum $\frac{n}{2^{i-1}+1}$ sites actifs. Pendant la phase i il y a donc au maximum $2 \cdot 2^i \cdot \frac{n}{2^{i-1}+1}$ messages qui transitent sur le réseau.
- Le nombre de phases est de $\log_2 n$ (au delà on "boucle"). Ainsi il y a moins de $\sum_{i=1}^{\log_2 n} 2 \cdot 2^i \cdot \frac{n}{2^{i-1}+1} \leq 4n \log_2 n$ échanges de messages pendant l'algorithme et ceci quelque soit la répartition des valeurs sur le cycle.
- Le gagnant est celui qui reçoit sa propre valeur à la phase $\log_2 n$ et qui ne reçoit pas d'autres valeurs plus grandes. Il peut alors proclamer qu'il est le gagnant.
NB : la complexité a été améliorée par rapport aux exercices et mais dans le cadre d'un cycle bi-directionnel (plus puissant donc).
- Voici l'algorithme :

```

Phase i
  Tant que (Etat == actif)
    Envoyer(< Vali, 2i >);
    i = i + 1;

Reception Message(< V, TTL >)
  Si (Etat == actif) et (TTL <> 1) alors
    Faire transiter le message
  Sinon
    Si V > vali alors etat = passif
    Sinon Si V = vali alors Elu;

```

TTL :time to live

un autre algorithme :

```

Reveil par signal      etat=actif
i = 0
Tant que (etat = actif)
  Envoyer (election, vali, 2i)
  ATTendre (reception messages)

```

Réception du message (*election, V, D*) sur j

```

si etat = actif
si V > Valj
  Alors etat = perdant ;
Si (V = Valj)
  etat = gagnant
Si etat = perdant
  Si D - 1 > 0
    Envoyer (election, V, D - 1)

```

- L'inconvénient de cet algorithme c'est que certains sites passifs recevront, en tant que destinataire, des messages qui ne servent à rien. Il vaut mieux envoyer à un site encore actif (voir l'exercice suivant).

Fin correction exercice 3

Exercice 4 – L'algorithme de Franklin L'algorithme proposé dans cet exercice est un raffinement du précédent. Le code pour un site i est :

```

Lors du réveil du site i
  etati := actif ;
  Tant que (etati = actif) faire
    Envoyer(vali) à droite ;
    Envoyer(vali) à gauche ;
    Attendre un message (Vd) venant de la droite ;
    Attendre un message (Vg) venant de la gauche ;
    V := max{Vd, Vg} ;
    Si (V ≥ vali) alors etati := passif ;
  Si (vali = V) alors
    gagnanti = i ;
    Prévenir les autres sites
  Sinon faire suivre tous les messages.

```

- En quoi cet algorithme est-il différent du précédent ? Peut-on remplacer \geq par $>$?
- Combien de messages sont générés ?

Correction exercice 4

Cette version permet de restreindre l'envoi des messages jusqu'au prochain actif au lieu de l'envoyer sur une distance donnée a priori. On ne peut pas supprimer \leq , dans le cas contraire on bouclerait. Du coup les messages ne contiennent plus le compteur de sauts et ils ne parcourent plus des distances inutiles. En effet, dans l'exercice précédent il y a envoi des messages à une distance 2^i pour un site actif. On remarque que cet algorithme marche, de manière implicite, par phase. A chaque phase au moins la moitié des sites deviennent passifs ; Il faut donc au plus $\log_2 n$ phases avant la

proclamation des résultats. Pendant chaque phase chaque lien est traversé une fois dans chaque sens. Ce qui conduit à $2n$ messages par phase. D'où une complexité totale d'au plus $2n \log_2 n$ messages. La terminaison est garantie lorsque le site gagnant reçoit sa propre valeur. On peut sortir du Tant que à ce moment là. Pour décider du vainqueur, il faut sortir de la boucle Tant que.

Fin correction exercice 4

Exercice 5 – Anneau Retour sur les cycles unidirectionnels. On propose l'algorithme suivant pour le site i .

Réveil spontané du site i ou réception d'un message pour la première fois

```
Envoyer(< 1, vali >);
etati := actif;
maxi = vali;
```

Lors de la réception du message < 1, V >

```
Si (etati = actif) alors
  Si (V ≠ maxi) alors
    Envoyer(< 2, V >);
    voisini := V;
  Sinon maxi est élu, prévenir les autres;
Sinon Envoyer(< 1, V >);
```

Lors de la réception du message < 2, V >

```
Si (etati = actif) alors
  Si (voisini > V) et (voisini > maxi) alors
    maxi := voisini;
    Envoyer(< 1, maxi >);
  Sinon etati := passif;
Sinon Envoyer(< 2, V >);
```

1. A quoi servent les variables locales max_i , $voisin_i$ et les deux messages? Par qui peut être proclamé le résultat?
2. Quelle est la complexité en nombre de messages échangés?
3. Appliquer l'algorithme sur le cycle donné par la figure 1.

Correction exercice 5

IL est important de noter que le gagnant peut-être passif Dans les algorithmes pour les cycles bi-directionnels, on pouvait savoir si on était un site ayant un maximum local. Si oui on continuait sinon on était perdant. Ici on ne peut pas connaître la valeur du suivant dans le cycle. Dans ce nouvel algorithme, un site i reste *actif* si son **prédécesseur (actif) est un maximum local** (sinon il devient *passif*); c'est à dire si i (lui même) et $i - 2$ (le prédécesseur de son prédécesseur)

ont chacun des valeurs plus petites que celle de $i - 1$ (prédécesseur de i). Pour cela il faut donc que i reçoive la valeur de $i - 1$ mais aussi celle de $i - 2$, d'où les deux messages. Les deux variables max_i et $voisin_i$ servent donc à enregistrer ces valeurs.

Lorsqu'un site reçoit une valeur qu'il avait déjà reçu (et enregistré dans la variable max_i) l'algorithme est terminé. Il connaît donc le gagnant et peut le proclamer.

Question 2. Ici encore il y a $\log_2 n$ phases naturelles. Or, durant chaque phase chaque lien est traversé par au plus deux messages (messages 1 et 2) d'où une complexité inférieure à $2n \log_2 n$ messages.

NB : on obtient donc aussi une complexité en $O(n \log n)$ même dans le cas d'un cycle unidirectionnel (en "émulant" en quelque sorte les algorithmes pour les cycles bi-directionnels).

Il est important de noter que le site qui détecte le vainqueur n'est pas forcément le vainqueur à la différence des premier algorithme.

Fin correction exercice 5

Exercice 6 – Election dans un arbre.

On suppose que l'on a un réseau d'ordinateurs connectés en *arbre* (voir exemple figure 3). Chaque canal est 'FIFO'. Les sommets (ou sites) ne communiquent directement qu'avec leurs *voisins* immédiats. Chaque sommet a un *identificateur* unique.

1. Ecrire un algorithme réparti qui :
 - "Réveille" tous les sommets à partir d'au moins un *initiateur*.
 - Calcule la plus petite étiquette de l'arbre (le gagnant étant le sommet qui possède cette étiquette). Pour cela :
 - Faire un parcours des feuilles vers l'intérieur de l'arbre en calculant la plus petite étiquette au fur et à mesure.
 - Propager le résultat de l'élection de voisin en voisin. Si un sommet reçoit son étiquette, il se déclare *vainqueur*, sinon *perdant*.

Pour cela vous pourrez utiliser les variables suivantes :

Un site p a les variables locales suivantes :

$veille_p$ booléen initialisé à *Faux*,
 $voisins_veilles_p$ entier initialisé à 0,
 rec_p tableau de $|voisins_p|$ booléens initialisés à *Faux*,
 val_p la valeur pour l'élection, v_p , initialisée à la valeur val_p ,
 $etat_p \in \{neutre, gagnant, perdant\}$, initialisé à *neutre*.

Vous devez commenter votre algorithme.

2. Proposez une exécution possible sur l'arbre de la figure 3.
3. Quelle est la complexité en nombre de messages échangés?
4. Pourquoi commencer par les feuilles?
5. Que faire si l'on perd l'hypothèse de l'identificateur unique?

Correction exercice 6

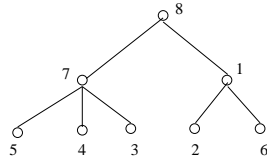


FIG. 3 – Un réseau en arbre.

Question 1. On supposera que l'arbre contient n sommets. Un site p a les variables locales suivantes :

$veille_p$ booléen initialisé à *Faux*,
 $voisins_veillees_p$ entier initialisé à 0,
 rec_p tableau de $|voisins_p|$ booléens initialisés à *Faux*,
 val_p la valeur pour l'élection, v_p , initialisée à la valeur val_p ,
 $etat_p \in \{neutre, gagnant, perdant\}$, initialisé à *neutre*.

Si (p est initiateur) alors

$veille_p := Vrai$;
 Pour tout ($q \in voisins_p$) faire
 Envoyer($\langle REVEIL \rangle$) à q ;

Lors de la réception d'un message pour la première fois

(** PHASE DE REVEIL : **)

Tant que ($voisins_veillees_p < |voisins_p|$) faire
 Attendre Recevoir message($\langle REVEIL \rangle$) ;
 $voisins_veillees_p := voisins_veillees_p + 1$;
 Si (non $veille_p$) alors
 $veille_p := Vrai$;
 Pour tout ($q \in voisins_p$) faire
 Envoyer($\langle REVEIL \rangle$) à q ;

(** PHASE DE CALCUL : **)

$v_p := val_p$;
 Tant que ($|\{q : non\ rec_p[q]\}| > 1$) faire
 Attendre Recevoir($\langle TOKEN, r \rangle$) de q ;
 $rec_p[q] := Vrai$;
 $v_p := \min\{v_p, r\}$;
 Envoyer($\langle TOKEN, v_p \rangle$) à q_0 tel que $rec_p[q_0] = Faux$;

Attendre Recevoir($\langle TOKEN, r \rangle$) de q_0 ;

$v_p := \min\{v_p, r\}$;
 Si ($v_p = val_p$) alors $etat_p := gagnant$;
 Sinon $etat_p := perdant$;
 Pour tout ($q \in voisins_p, q \neq q_0$) faire
 Envoyer($\langle TOKEN, v_p \rangle$) à q ;

L'algorithme consiste à réveiller tous les sommets à partir d'un (ou plusieurs) initiateur. Ensuite,

à cause du premier **Tant que** (de la phase de calcul), les feuilles envoient leur valeur. Un sommet interne de l'arbre envoie sa valeur au seul sommet qui ne lui a rien envoyé (c'est la condition stricte). A chaque étape, la valeur minimale est calculée. La dernière partie consiste à propager la valeur minimale dans tous le réseau.

Question 2. Nombre de messages échangés. Pendant la phase de réveil : deux messages $\langle REVEIL \rangle$ et deux messages $\langle TOKEN \rangle$ sont envoyés sur chaque lien. En tout donc 4 messages par lien. D'où $4n - 4$ messages en tout. **Fin correction exercice 6.**

Exercice 7 – Borne inférieure pour un réseau quelconque

Nous considérons un graphe quelconque. Nous cherchons à déterminer une borne inférieure pour tout algorithme d'élection basé sur la comparaison des identités. A votre avis quelle est cette borne? **Aide :** Aidez-vous des résultats obtenu sur l'anneau.

Correction exercice 7

Nous avons le résultat suivant : Pour n'importe quel algorithme basé sur la comparaison des identités pour n'importe quel réseau possède (dans le pire des cas et dans le cas moyen) une complexité d'au moins de $\Omega(|E| + n \log n)$.

En effet, la borne $\Omega(n \log n)$ est une borne inférieure car les réseaux quelconques incluent les anneaux pour lesquels $\Omega(n \log n)$ est une borne inférieure. Pour voir que $|E|$ messages est également une borne inférieure, même dans le cas des meilleurs complexité, supposons qu'il existe un algorithme d'élection A possède une complexité sur un réseau G pour lequel nécessite moins de $|E|$ messages échangés. Construisons un réseau G' en connectant deux copies de G en rajoutant une arête, entre ces deux copies, à partir d'un sommet rajouté sur une arête non visité par l'algorithme A (nous avons que cette arête existe car le l'algorithme utilise moins de $|E|$ messages). Les identités dans chaque partie du réseau admettent le ordre relatif que dans G . La complexité du problème d'élection dans G peut être simulé dans les deux partie de G' , entrainant une élection dans chaque partie de G' .

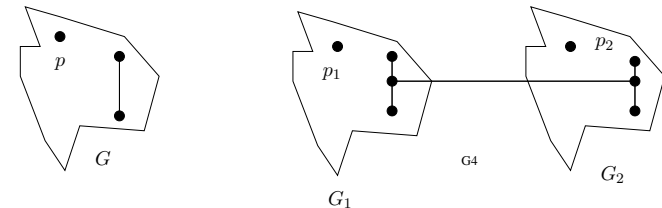


FIG. 4 – A calcul avec deux leaders

Fin correction exercice 7

Exercice 8 – Un algorithme asynchrone dans un réseau complet : algorithme de Humblet

Dans le but de développer des algorithmes efficaces dans le cas d'un graphe complet, nous allons utiliser la technique d'acquisition de territoire.

Dans la procédure d'acquisition chaque candidat essaie de capturer un voisin à chaque itération ; en utilisant un message du type *capture* contenant son identité et le nombre de sommets déjà capturés avec la variable $stage(x)$. Si la tentative est un succès, le voisin attaqué devient capturé, et le sommet candidat entre dans une nouvelle itération sinon le candidat devient passif. Un sommet capturé retient l'identité, et la valeur de la variable $stage(x)$ et le lien de son maître (l'identité de celui qui l'a capturé).

1. Remplir les spécifications suivantes :

Nous supposons qu'un candidat x envoie un message du type *capture* à y :

- (a) Si y est candidat :
 - i. Si $stage(x) > stage(y)$, alors conclure sur l'attaque.
 - ii. Si $stage(x) = stage(y)$, donner les états possibles pour x et y .
 - iii. Si $stage(x) < stage(y)$, alors conclure sur les états de x et de y .
 - (b) Si y est passif, conclure sur le résultat de l'attaque de x sur y .
 - (c) Si y est déjà capturé alors x doit défaire le maître de y , noté z avant de capturer y . Dans ce cadre y envoie un message du type Warning à z avec l'identité de x et $stage(x)$.
 - i. Si z est candidat. Donner les différents cas de figures.
 - ii. Dans le cas contraire donner tous les cas de figures.
 - (d) Si l'attaque est un succès, conclure sur les états de x et y sur la valeur de $stage(x)$.
2. Donner l'algorithme maintenant.
 3. Applique l'algorithme de Humblet sur le graphe donné par la figure 5.

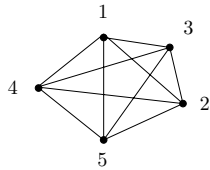


FIG. 5 – Graphe complet à 5 sommets

4. Montrer que l'algorithme répond positivement au problème de l'élection.
5. Que se passe-t-il quand un sommet a capturé au moins $n/2 + 1$ sommets ?
6. Combien de sommets peuvent être candidat à l'itération i ? Déterminer la taille maximum d'un territoire à l'itération i ? Les territoires sont-ils disjoints ?
7. Combien de messages au maximum sont nécessaires lorsque un sommet procède à une demande de capture ?
8. Combien de stages sont nécessaires pour la terminaison de l'algorithme ?

9. Conclure sur la complexité en nombre de messages.
10. Donner la complexité en temps.

Correction exercice 8

1. Remplir les spécifications suivantes :

Nous supposons qu'un candidat x envoie un message du type *capture* à y :

- (a) Si y est candidat :
 - i. Si $stage(x) > stage(y)$, alors l'attaque est un succès.
 - ii. Si $stage(x) = stage(y)$, alors l'attaque est un succès si $id(x) < id(y)$, sinon x devient passif.
 - iii. Si $stage(x) < stage(y)$, alors x devient passif.
- (b) Si y est passif, alors l'attaque est un succès.
- (c) Si y est déjà capturé alors x doit défaire le maître de y , noté z avant de capturer y . Dans ce cadre y envoie un message du type Warning à z avec l'identité de x et $stage(x)$.
 - i. Si z est candidat. Si $stage(z)$ est plus grand que $stage(x)$, ou $stage(x) = stage(z)$ et $id(z) < id(x)$, alors l'attaque sur y est un échec. z notifie à x son échec via y .
 - ii. Dans les autres cas (z est déjà passif ou capturé, z est candidat avec un territoire plus faible que x , ou même stage mais avec une identité plus importante que x , l'attaque sur y est un succès. z notifie à x via y et si z était candidat alors il devient passif.
- (d) Si l'attaque est un succès, y est capturé par x , x incrémente $stage(x)$ et procède à une nouvelle itération.

Il est important de noter que chaque tentative de capture pour un sommet candidat coûte exactement deux messages (un pour le message *capture* et l'autre pour sa notification) si le voisin est également candidat ou passif. Dans le cas contraire si le sommet a déjà été capturé, deux autres messages additionnels seront échangés.

Nous pouvons optimiser l'utilisation du nombre de messages en bloquant les messages en y pour z n'ayant aucune chance de capturer z . Ainsi le sommet y transmettra un message du type warning à z et attend le retour de z , tout en stockant les messages venant d'autres sommets t ayant un stage plus grand et/ou une identité plus petite. Si l'attaque est un succès pour x alors y change de maître et envoie les messages de la file d'attente à x . Avec ce changement, il y a une exclusion mutuelle sur les territoires pour deux sommets candidats.

2. Voici l'algorithme

```

ETAT := endormi En cas de réveil spontané
stagei := 1 ; value := id(x)
Voisinsi := N(x)
next = head(voisinsi) ;
envoyer(capture, stagei, valuei) à next
etat := candidate
    
```

ETAT : endormi En cas de réception de (capture, $stage_x$, $valeur_x$) de x
envoyer(acceptation, $stage_x$, $valeur_x$) à x
 $stage_i = 1$; $maitre_i := x$
 $stage_{dumatre} := stage_x + 1$

ETAT : candidat En cas de réception de (capture, $stage_x$, $valeur_x$) de x
Si $stage_x < stage_i$ ou ($stage_x = stage_i$ et $valeur_x > valeur_i$) alors envoyer(échec, $stage_i$)
à x Sinon envoyer(acceptation, $stage_x$, $valeur_x$) à x
 $maitre_i := x$
 $stage_{dumatre} := stage_x + 1$
etat :=capturé

ETAT : candidat En cas de réception de (acceptation, $stage_x$, $valeur_x$) de x
 $stage_i++$ Si $stage_i \geq 1+n/2$ alors envoyer(terminaison) à $voisin_i$ Sinon $next =$
 $Head(voisin_i)$ envoyer(capture, $stage_i$, $valeur_i$) à next

ETAT : candidat En cas de réception de (échec, $stage_x$) de x
etat=passif

ETAT : candidat En cas de réception de (alerte, $stage_x$, $valeur_x$) de y
Si $stage_x < stage_i$ ou ($stage_x = stage_i$ et $valeur_x > valeur_i$) alors envoyer(non, $stage_i$)
à y Sinon envoyer(oui, $valeur_x$) à y
etat :=passif

ETAT : passif En cas de réception de (capture, $stage_x$, $valeur_x$) de x
Si $stage_x < stage_i$ ou ($stage_x = stage_i$ et $valeur_x > valeur_i$) alors envoyer(échec, $stage_i$)
à x Sinon envoyer(acceptation, $stage_x$, $valeur_x$) à x
 $maitre_i := x$
 $stage_{dumatre} := stage_x + 1$
etat :=capturé

ETAT : passif En cas de réception de (alerte, $stage_x$, $valeur_x$) de y
Si $stage_x < stage_i$ ou ($stage_x = stage_i$ et $valeur_x > valeur_i$) alors envoyer(non, $stage_i$)
à y Sinon envoyer(oui, $valeur_x$) à y
etat :=passif

ETAT : capturé En cas de réception de (capture, $stage_x$, $valeur_x$) de x
Si $stage_x < stage_{maitre}$
envoyer(échec, $stage_{maitre}$) à x Sinon $attaquant_i = x$
envoyer(alerte, $stage_x$, $valeur_x$) à $maitre_i$
enfile($voisin_i - maitre_i$)
etat :=capturé

ETAT : capturé
En cas de réception de (non, $stage_y$) de x

$Open(voisin_i)$
envoyer(échec, $stage_x$) à attaquant
Sinon
ETAT : capturé
En cas de réception de (oui, $stage_y$) de x
 $stage_{maitre} = stage_x + 1$
envoyer(échec, $stage_x$) à attaquant
 $Open(voisin_i)$
envoyer(acceptation, $stage_x$, $valeur_x$) à attaquant

ETAT : capturé
En cas de réception de (alerte, $stage_y$, $valeur_y$) de x
Si $stage_y < stage_{maitre}$ alors
envoyer(non, $stage_{maitre}$) à x
Sinon
envoyer(oui, $stage_x$, $valeur_x$) à x

3. L'application
4. A la fin de l'algorithme, il nous reste un seul sommet en lice (il suffit de supposer que deux sites sont élus, et ceci est impossible), donc c'est l'élu. Il est important de noter que l'élu n'est pas forcément celui qui a la plus petite identité.
5. L'algorithme se termine car aucun autres sommets ne peut avoir un territoire supérieur à celui qui a un territoire de taille $n/2 + 1$. Nous avons prouver la terminaison.
6. Les territoires de tailles i sont disjoints, ainsi ils ne peuvent pas être plus grand que $n_i \leq n/i$ à l'itération i .
7. Il existe 4 messages au maximum.
8. Il y a $n/2 + 1$ itérations.
9. En rajoutant $n - 1$ messages pour avertir les voisins, nous trouvons :
$$n - 1 + \sum_{i=1}^{n/2+1} 4n_i \leq n + 1 + \sum_{i=1}^{n/2} 4n = 4nH_{n/2} + n + 1$$
10. La complexité en temps est $O(n)$.

Fin correction exercice 8

Exercice 9 – Algorithme de Chan-Chin pour l'élection synchrone dans un graphe complet

Le réseau est ici supposé totalement synchrone : une horloge globale est accessible à tout processus du réseau dont chacune des actions est synchronisée cycle par cycle ; un message envoyé au tems t est donc assuré d'une réception au temps $t + 1$ / Les processus peuvent devenir actifs et déclencher l'exécution de l'algorithme d'élection à tout instant, soit spontanément, soit à la suite de la réception d'un message.

Chaque processus possède trois variables locales id , $etat$, $phase$:

- id : identité courante du détenteur du processus considéré et à l'initialisation $id = id_i$ du processus P_i .
- un processus actif admet deux états : *candidat* ou *détenu*. Un candidat est son propre détenteur et cherche à détenir les autres pour être élu ; dans le cas contraire être *détenu* signifie que nous ne poursuivons pas le processus d'élection. Un site peut changer plusieurs fois de détenteur mais appartient à un seul détenteur.

L'algorithme est fondé sur le principe suivant : l'unique processus qui finit par détenir tous les autres est le processus élu. *phase* est initialisé à 0 et est incrémentée d'une unité à chaque cycle pair de l'horloge globale ; la variable *phase* mesure le nombre total de processus dont le détenteur est le même que pour le processus courant considéré. Deux catégories de messages peuvent être échangés, les messages (*phase, id*) et les messages *purge*.

A chaque incrémentation de phase, au début de tout cycle pair d'horloge, un candidat tente de doubler le nombre de ses détenus en envoyant son message (*phase, id*) le long $2^{phase-1}$ lignes de communications non encore marquées. Ces lignes sont alors marquées pour ce processus. Un candidat ne peut continuer comme tel au cycle pair suivant que s'il n'a reçu aucun message (*purge*) entre-temps, sinon il devient détenu. En cas de succès i.e. aucun message *purge* reçu, le candidat finira par être le seul détenteur de 2^{phase} processus.

Au début de chaque cycle impair, chaque processus P considère son couple maximal (du point de vue lexicographique) ($phase^*, id^*$) déjà reçu. En réponse à tout autre message du typ (*phase, id*), il renvoie un message *purge*. Si le couple maximal reçu ($phase^*, id^*$) est inférieur au couple courant (*phase, id*) du processus P considéré, ce dernier envoie aussi un message *purge*. Sinon, le processus d'origine du couple maximal ($phase^*, id^*$) devient détenteur du processus P : le couple (*phase, id*) de P devient le couple maximal ($phase^*, id^*$), P devient ou reste détenu et un message *purge* est envoyé au précédent détenteur de P . En d'autres termes, tout processus devenant ou restant détenu ne conserve qu'un seul et unique détenteur.

Un processus arrête la procédure d'élection lorsqu'il découvre que son détenteur détient aussi tous les autres processus du réseau. L'élection est donc terminée.

1. Donner l'algorithme.
2. Donner l'exécution de l'algorithme sur la graphe de la figure 5.
3. Donner la complexité dans le pire des cas en nombre de messages.
4. Donner la complexité en temps de l'algorithme.
5. Calculons maintenant la complexité en moyenne en messages :
 - (a) Soit α le nombre de processus qui est détenu de manière disjointe par chaque candidat au début de la phase j et β le nombre de messages envoyés par chaque en début de phase j . Le nombre moyen de candidat en début de phase $j + 1$ sera au plus de $1 + \frac{\alpha}{\alpha\beta}$
 - (b) Donner le nombre moyen de candidats au début de la phase 1. Au début de la phase 2, et en début de phase $j + 1$? Montrer que la borne supérieure est $7n$.
 - (c) Conclure sur l'optimalité en moyenne.

Correction exercice 9

- 1.
- 2.
3. Calculons maintenant la complexité en moyenne :
 - (a) le nombre moyen de candidats en début de phase 1 est au plus n . Pour le début de la phase 2 il est au plus $n/2$. Grâce à l'égalité $\alpha = \beta = 2^{j-1}$ et d'après la question précédente, il y a au plus $1 + n/(2^{2j-2})$ candidats en début de phase $j + 1$ ($j = 2, \dots, \lceil \log n \rceil - 1$). Le nombre de messages (*phase, id*) envoyés par chaque candidat à chaque incrémentation de phase est $2^{phase-1}$; donc en phase $j + 1$, chaque envoi 2^j messages (*phase, id*). Comme pour chaque message (*phase, id*), il y a au plus message *purge*, le nombre moyen de messages est majoré par

$$= 2n + 2^2 \cdot n/2 + \sum_{j=2}^{\lceil \log n \rceil - 1} (1 + \frac{n}{2^{2j-2}}) 2^{j+1} \quad (1)$$

$$= 4n + \sum_{j=3}^{\lceil \log n \rceil} 2^j + \sum_{j=1}^{\lceil \log n \rceil - 2} \frac{n}{2^j} \leq 7n \quad (2)$$
 - (b)
 - (c) La borne inférieure de la complexité en messages de l'élection sur un réseau complet est clairement en $\omega(n)$: l'algorithme de Chan-Chin est donc optimal en moyenne de complexité en message $\theta(n)$.

Fin correction exercice 9

Exercice 10 – Anneau unidirectionnel, algorithme de Peterson/Dolev-Klawe-Rodeh

L'algorithme de Chang-Roberts admet une complexité de $O(n \log n)$ en nombre d'échange de messages en moyenne, mais pas dans le pire des cas. Un algorithme avec une complexité dans le pire des cas avec $O(n \log n)$ a été proposé par Franklin, mais cet algorithme nécessite que les canaux soient bi-directionnels. Peterson et Dolev/Klawe/Rodeh ont proposé de manière indépendante un algorithme similaire utilisant seulement $O(n \log n)$ messages dans le pire des cas pour les anneaux. L'algorithme calcule dans un premier temps la plus petite identité et la fait connaître à tous les processus, et le processus ayant l'identité la plus petite valeur devient le leader et les autres non. Initialement chaque identité est active, mais à chaque tour des identités deviennent passives. Dans un tour une identité active compare sa propre valeur avec ces deux voisins ayant une identité active dans le sens des aiguilles d'une montre (resp. dans le sens inverse). Si nous avons un minimum local, l'identité survit au tour, sinon l'identité devient passive. Sachant que toutes les identités sont différentes, la prochaine identité ayant le minimum local n'est plus un minimum local, ce qui implique au moins la moitié des identités de survivent pas. En conséquence, après $\log n$ tours, il ne reste plus qu'une seule identité, et c'est le gagnant. L'algorithme pour les anneaux unidirectionnel et donné par l'algorithme 1. Le processus p est actif dans un tour, et admet l'identité c_p au début du tour. Sinon, p est passif et relai simplement tous les messages qu'ils reçoient. Un processus encore actif envoie son identité courante au prochain processus actif, et obtient l'identité courante d'un précédent processus encore actif en utilisant le message $\langle one, . \rangle$. L'identité reçu est stocker (dans

la variable acn_p) et pour les identités survivantes après le tour admette l'identité courante de p au prochain tour. Pour déterminer si l'identité acn_p existe au tour, elle devra être comparé avec ci_p et avec l'identité obtenu par le message $\langle two, . \rangle$. Le processe p envoie le message $\langle two, acn_p \rangle$ pour effectuer une possible décision au prochain processe actif. Une exception apparait quand $acn_p = ci_p$, et dans ce cas l'identité reste active et annoncera à tous les autres processus en utilisant le message $\langle smal, acn_p \rangle$.

1. Donner les états initiaux des variables définit précédemment.
2. Compléter les parties manquantes.
3. Donner une exécution sur le graphe donné par la figure 1.
4. Expliquer pourquoi les liens doivent être FIFO.
5. Montrer que dans le pire des cas la complexité de cet algorithm est en $O(n \log n)$.
 - (a) Montrer que si une boucle i commence avec $k > 1$ sites actifs, et chaque site possède un ci_p différent, alors au moins un et au plus $k/2$ sites survivent dans la boucle. A la fin d'un tour de boucle toutes les identités courantes des sites actifs sont différentes et contient la plus petite identité.
 - (b) Conclure.
 - (c) Montrer que si l'algorithme commence avec un site actif p , avec l'identifiant courant ci_p , l'algorithme termine quand $win_p = acn_p$ for chaque q .
 - (d) Quel est la condition d'arrêt.
 - (e) Donner le nombre maximum de tour dans la boucle.

Correction exercice 10

1. $ci_p = p, acn_p = undef, win_p = undef, state_p \in \{active, passive, leader, lost\}$ et $state_p = active$;
2. Voici l'algorithme complété.
3. Il est clair que si les canaux ne sont pas fifos, l'algorithme n'est pas valable.
4. On va démontrer que la complexité est en $O(n \log n)$.
 - (a) Par les échanges de messages $\langle one, q \rangle$, entre les sites passifs, chaque site actif obtien l'identité courante de son premier voisin actif, ce qui diffèrent dans tous les cas de sa propre identité. Par conséquence, chaque site actif continue dans la boucle avec les échanges de messages $\langle two, q \rangle$, par lequel chaque site actif obtiennent également l'identité courante du second voisin actif. Chaque site actif possède une valeur différente de acn , impliquant les sites survivants de la boucle ont une valeur d'identité différentes à la fin de celle-ci. Au moins la plus petite identité au début du tour survit, ainsi il y a au moins un survivant. La prochaine identité du minimum local n'est pas un minimum local, impliquant que le nombre de survivants est au plus $k/2$.
 - (b) La proposition précédente implique qu'un tour avec $\leq \lfloor \log n \rfloor$ qui commence avec exactement une identité active c'est à dire la plus petite des tous les initiateurs.

Algorithm 1 Algorithm de Peterson/Dolev-klawe-Rodeh

```

var :  $ci_p, acn_p, win_p, state_p$ 
if  $p$  est initiateur then
    .....
else
    .....
end if
while  $win_p = undef$  do
    if  $state_p = active$  then
        ...
        ...
         $acn_p := q$ ;
        if  $acn_p = ci_p$  then
            {*( $acn_p$  est le minimum)*}
            send  $\langle smal, acn_p \rangle$ ;
             $win_p = acn_p$ 
            receive  $\langle smal, q \rangle$ 
        else
            {* $acn_p$  admet l'identifiant courant de son voisin*}
            send  $\langle two, acn_p \rangle$ ;
            receive  $\langle two, q \rangle$ ;
            if  $acn_p < ci_p$  &  $acn_p < q$  then
                 $ci_p := acn_p$ 
            else
                 $state_p := passive$ ;
            end if
        end if
    end if
else
    {* $state_p = passive$ *}
    ...
    ...
    ...
    {* $m$  est soit du type  $\langle two, q \rangle$  ou soit  $\langle smal, q \rangle$ *}
    if  $m$  est un message du type  $\langle smal, q \rangle$  then
         $win_p := q$ 
    end if
    end if
end while
if  $p = win_p$  then
     $state_p := leader$ 
else
     $state_p := lost$ 
end if

```

Algorithm 2 Algorithm de Peterson/Dolev-klawe-Rodeh

```
var :  $ci_p, acn_p, win_p, state_p$ 
if  $p$  est initiateur then
   $state_p := active$ 
else
   $state_p := passive$ ;
end if
while  $win_p = undef$  do
  if  $state_p = active$  then
    Envoie  $\langle one, ci_p \rangle$ ;
    receive  $\langle one, q \rangle$ ;
     $acn_p := q$ ;
    if  $acn_p = ci_p$  then
       $\{*(acn_p \text{ est le minimum})*\}$ 
      send  $\langle smal, acn_p \rangle$ ;
       $win_p = acn_p$ 
      receive  $\langle smal, q \rangle$ 
    else
       $\{*acn_p \text{ admet l'identifiant courant de son voisin}*\}$ 
      send  $\langle two, acn_p \rangle$ ;
      receive  $\langle two, q \rangle$ ;
      if  $acn_p < ci_p \ \& \ acn_p < q$  then
         $ci_p := acn_p$ 
      else
         $state_p := passive$ ;
      end if
    end if
  else
     $\{*state_p = passive*\}$ 
    receive  $\langle one, q \rangle$ ;
    send  $\langle one, q \rangle$ ;
    receive  $m$ ;
    send  $m$ ;
     $\{*m \text{ est soit du type } \langle two, q \rangle \text{ ou soit } \langle smal, q \rangle*\}$ 
    if  $m$  est un message du type  $\langle smal, q \rangle$  then
       $win_p := q$ 
    end if
  end if
end while
if  $p = win_p$  then
   $state_p := leader$ 
else
   $state_p := lost$ 
end if
```

- (c) Le message $\langle one, ci_p \rangle$ de p est relayé par tous les sites et finalement reçu par p . Le site p obtient donc $acn_p = ci_p$ et envoie un message $\langle smal, acn_p \rangle$ sur l'anneau, impliquant la sortie des process q de la boucle principale avec $win_p = acn_p$.
- (d) L'algorithme termine quand tous les sites sont d'accord sur l'identité du leader (dans la variable win_q); tous les sites perdants sont dans l'état perdant et le vainqueur dans l'état leader.
- (e) Au maximum il y a au plus $\lceil \log n \rceil + 1$ dans lequel $2n$ sont échanger ce qui prouve que la complexité $2n \log n + O(n)$

Fin correction exercice 10

Algorithme distribué
TD – Séance n° 2

Exercice 1 – L’algorithme de Ricart-Agrawala

Dans l’algorithme suivant, chaque site désirant la ressource va demander la permission à tous les autres. On départage les conflits en étiquetant chaque demande par l’heure (logique) à laquelle on a fait cette demande. Les demandes les plus anciennes sont les plus prioritaires.

Procédure acquisition

```

demandeuri := vrai ;
horlogei := horlogei + 1; heure_demandei := horlogei;
rep_attenduesi := n - 1 ;
Pour tout (xi ∈ V - {i}) faire
    Envoyer(< REQUEST, heure_demandei >) à xi ;
Attendre(rep_attenduesi = 0) ;
    
```

Procédure libération

...

Procédure reception de request

...

Lors de la réception de < REPONSE > de j

```
rep_attenduesi := rep_attenduesi - 1 ;
```

1. Donner le rôle des variables $heure_demande_i$, $rep_attendues_i$, $demandeur_i$ et $differe_i$.
2. Donner la procédure de libération et de reception.
3. Donner la complexité en nombre de messages.
4. Montrer que l’algorithme de Ricart-Agrawala respecte le principe de sûreté c’est à dire que il existe au plus un site en section critique.
5. Montrer que l’algorithme de Ricart-Agrawala respecte le principe de vivacité c’est à dire que qu’un site pourra toujours rentrer en section critique.
6. Donner l’intervalle de temps de non utilisation de la section critique sachant que les messages ont un temps de transit borné par T .

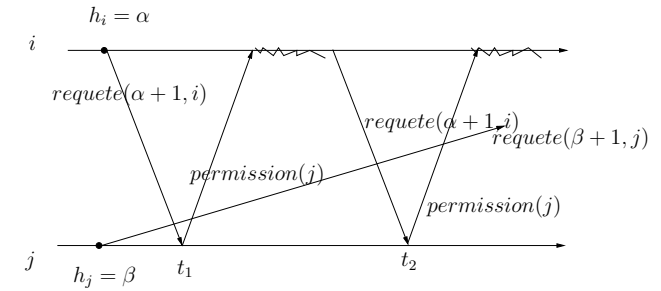


FIG. 1 – Effets du déséquencement

7. Avons-nous utilisé pour la démonstration que les canaux sont fifos ou non? Une question se pose : Quelle incidence peut avoir le caractère non fifos des canaux sur le comportement de l’algorithme? Pour répondre à cela nous allons nous limiter à 2 sites i et j qui obéissent au comportement suivant :

Boucler sur Acquérir ;

```
< utilisation de la section critique >
```

librer ;

Fin de boucle

et nous supposons que $h_i = \alpha$ et $h_j = \beta$ avec $(\alpha, i) < (\beta, j)$. Une exécution possible est décrite par la figure 1 (la zone hachurée indique que le site correspondant est en section critique). Dans ce scénario les deux sites ont initialement fait des requêtes estampillées respectivement $(\alpha + 1, i)$ et $(\beta + 1, j)$ en t_1 . A la reception de la requête, le site j renvoie sa permission qui double la requête qu’il avait envoyée précédemment. A la réception du message *permission*, le site i rentre en section critique, puis en sort et invoque à nouveau *acquérir*.

- (a) Quel est la valeur de l’horloge h_i à cet instant t_2 (que vaut x ? Quel conséquence?
- (b) Que se passe-t-il concernant l’horloge de i quand le site i reçoit la requête de j ?
- (c) Un site peut-il rentrer un nombre de fois arbitraire en section critique alors qu’un autre site en a fait la demande? Ce nombre peut-il être non borné?
- (d) Donner une exécution dans le cas où les canaux seraient fifos.
- (e) Comment obtenir, avec des canaux non fifos, un ordre de satisfaction des demandes analogues à celui que l’on obtiendrait avec des canaux fifos sans utiliser des messages supplémentaires?

8. Quel est le défaut de cet algorithme concernant les demandes de permissions.
9. On suppose maintenant que le temps de transfert est connu. Soit δ le majorant sur les temps de transfert entre deux sites.
 - (a) Modifier l'algorithme de Ricart-Agrawala pour prendre en compte cette nouvelle hypothèse.
 - (b) Evaluer le nombre de messages par utilisation de la section critique.
10. On a fait l'hypothèse implicite d'un maillage complet pour les communications c'est-à-dire d'un canal bidirectionnel entre tout couple de sites. Comment adapter l'algorithme si le maillage entre les sites est un anneau unidirectionnel? Montrer que le nombre de messages pour une utilisation de la section critique est $\frac{n(n+1)}{2}$. Proposer une idée afin que le nombre de messages soit n pour une utilisation de la section critique.
11. La valeur des horloges peut croître indéfiniment. Donner l'intervalle de temps entre deux demandes pour un même site i . Soit h_i l'horloge d'un site i . Quelle est la valeur de l'intervalle des horloges des autres sites? Quel est la longueur de l'intervalle. Comment éviter la dérive des horloges?

Correction exercice 1

1. Chaque site i maintient les variables locales suivantes. $heure_demande_i$ représente l'heure à laquelle i a fait sa dernière demande. $rep_attendues_i$ est un entier qui compte le nombre de réponses attendues par i suite à sa demande.
 $demandeur_i$ booléen qui est vrai si et seulement si i est demandeur de la ressource; initialisé à *faux*.
 $differe_i$ tableau de n cases de booléens. $differe_i[j] = vrai$ si et seulement si i a différé sa réponse à la demande de j ; initialisé à *faux*.

2. Procédure libération

```

demandeur_i := faux;
Pour tout (x_i ∈ V - {i}) faire
  Si (differe_i[x_i]) alors
    Envoyer(< REponse >) à x_i;
    differe_i[x_i] := faux;
et Lors de la réception de < REQUEST, h > de j
  horloge_i := max{horloge_i, h};
(2) Si (non demandeur_i ou (heure_demande_i, i) > (h, j)) alors

```

```

Envoyer(< REponse >) à j;
Sinon differe_i[j] := vrai;

```

- on fait le max pour resynchroniser les horloges, ceci garantit que toutes les requêtes futures du site i auront des dates supérieures à h . l'incrémentatation pour acquérir et (2) garantit une progression correcte du temps logique.
3. La complexité est $2(n - 1)$ messages par demande.
 NB : la variable $horloge_i$ est gérée implicitement ici grâce au sous-protocole présenté précédemment.
 4. Il s'agit de montrer qu'il y a, à tout instant, au plus un site en section critique. Pour démontrer cela raisonnons par l'absurde en supposant qu'il y a deux sites i et j simultanément en section critique. Si tel est le cas d'une part i a envoyé à j un message $reque(h, i)$ et a reçu $permission(j)$ et d'autre part j a envoyé à i un message $reque(k, j)$ et a reçu $permission(i)$. Pour en arriver là trois cas sont a priori possibles.
 - (a) le site i a envoyé sa requête à j et celui-ci l'a reçu avant de faire la sienne.
 Dans ce cas, à la reception du message de requête, le site j est dans l'état *dehors* : il renvoie sa permission à i et l'on a alors $h_j \geq h$. Lorsque le site j exécute *acquérir* et envoie sa requête on a : $k = last_j = h_j + 1$ et donc $k > h$. A la reception du message $reque(k, j)$ le site i se trouvera donc prioritaire et ne renverra donc pas sa réponse. Cette situation n'a donc pas pu conduire à violer l'exclusion mutuelle.
 - (b) le second cas a priori possible est obtenu en intervertissant les rôles de i et j .
 - (c) Le troisième cas est celui où deux sites ont déjà envoyé leurs requêtes quand ils en reçoivent. Dans ce cas chacun des sites est, à la reception de la requête, dans l'état *demandeur* et, comme l'on a soit $(h, i) < (k, j)$ soit le contraire, un seul des deux booléens *priorit* peut prendre la valeur vrai; Un seul des deux sites renverra donc sa permission.

Aucune n'est trois situations a priori possibles ne peut donc mettre en défaut la propriété de sûreté.

5. Les demandes de section critique sont totalement ordonnées par leurs estampilles; considérons celle qui, à un instant donné, est dotée de la plus petite soit (h, i) . Lorsque les messages de requêtes correspondant à cette demande arrivent sur les autres sites, ceux-ci ne peuvent être prioritaires et renvoient donc leurs permissions attendues et entrera alors en section critique. Comme une fois la section critique utilisée, la demande correspondante sort du système, que toutes les utilisations sont par hypothèse de durée finie et que les estampilles sont totalement ordonnées, toute demande finira par avoir la plus petite estampille et donc par être honorée.

6. En ce qui concerne le temps, le but est d'utiliser au maximum la section critique. On va donc compter la durée durant laquelle la section critique n'est pas utilisée bien qu'elle soit demandée. On suppose pour cela que les messages ont un temps de transit borné par T . Le pire des cas se produit lorsque un site désire utiliser la section critique et que celle-ci est libre. Ce site envoie donc des requêtes et reçoit les permissions par retour de courrier : soit $2T$. Si par contre la section critique est utilisée, les messages de requêtes transitent durant cette période ainsi qu'un certain nombre de permissions : au mieux un site reçoit toutes les permissions sauf celle de l'utilisateur de la section critique ; lorsque celui-ci la libère il lui renvoie sa permission : dans ce cas la période de non utilisation est égale à T .
7. Sur le problème des canaux fifos et non fifos.

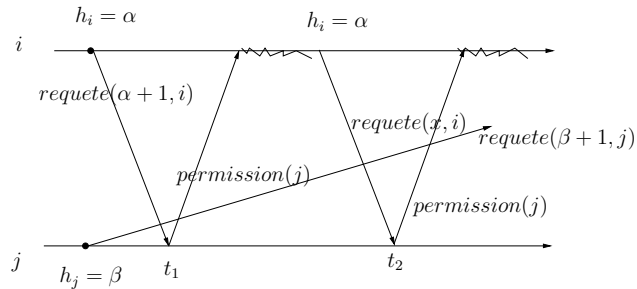


FIG. 2 – Effets du déséquencement

- (a) La valeur de l'horloge du site i au moment où il redemande à réentrer en section critique vaut toujours α car aucune requête n'a été reçue par i . La seconde requête du site i transporte donc la même estampille que précédemment en t_2 le site j renverra sa permission, et ainsi de suite..
- (b) Ceci peut se produire tant que la requête de j estampillé $(\beta + 1, j)$ n'est pas parvenue au site i ; lorsqu'elle lui parviendra celui-ci mettra à jour $h_i = \max\{\alpha, \beta + 1\}$ et dès lors les requêtes ultérieures de i auront des estampilles supérieures à $(\beta + 1, j)$.
- (c) En d'autres termes si les canaux ne sont pas fifos, il est possible qu'un site rentre en section critique un nombre arbitraire de fois alors qu'un autre site a effectué une demande. (Toutefois ce nombre est fini car les messages de requêtes finissent par arriver).

- (d) Si dans l'exemple, les canaux étaient fifos les messages $permission(j)$ ne pourraient doubler le message $requete(\beta + 1, j)$. Ce message serait le premier reçu par i qui, lors de cette réception, recalerait son horloge h_i à $\beta + 1$; et la seconde requêtes du site i , la requête du site j serait satisfaite puisque :

$$(\alpha + 1, i) < (\beta + 1, j) < (\beta + 2, i)$$

- (e) Pour cela il faut, sur tout le canal, faire transporter par les messages qui peuvent doubler les requêtes, une date plus grande ou égale à celle de la dernière requête envoyée sur ce canal. Il faut estampiller les messages $permissions$ envoyés de i vers j avec le couple $(\max(last_i, h_i), i)$, et recalculer les horloges logiques à la réception de tout message $requete$ ou $permission$.

8. le défaut est qu'un site voulant aller plusieurs fois en section critique devra demander l'autorisation à tous les autres sites même à ceux qu'ils ne veulent y rentrer.
9. Si l'on connaît un majorant δ sur les temps de transfert entre deux sites on peut modifier l'algorithme de la façon suivante. Après avoir diffusé ses requêtes un site i attend 2δ unités de temps. A la réception d'une telle requête un site j qui n'est pas prioritaire donne sa permission de façon implicite en ne renvoyant aucun message ! Si par contre j est prioritaire il donne par retour de courrier un réponse négative et renverra sa permission à sa sortie de section critique. Si le site i qui a diffusé ses requêtes à l'instant t ne reçoit pas de réponse d'ici $t + 2\delta$ il peut pénétrer en section critique ; dans le cas contraire il doit attendre les permissions qui viendront annuler les réponses négatives reçues. Le nombre de messages par utilisation de la section critique varie entre $n - 1$ et $3(n - 1)$. Cette version de l'algorithme n'est intéressante que si δ étant connu, est assez petit et s'il y a peu de conflits (dans ce cas en effet le nombre moyen de messages est proche de $n - 1$).

$NbRefus = 0$;

ProcédureAcquisition :

demandeur(i) = vrai;

horloge(i) ++;

heuredemande(i) = horloge(i);

Pour x_i dans V faire

envoyer($\langle Request, heuredemande(i) \rangle$);

fin pour

Attendre($2*\delta$)

Si $NbRefus == 0$

```

        SectionCritique() ;
    sinon
        Attendre(NbRefus == 0) ;
    Fin Si

ProcédureRéceptionReponse :
    Si message.Type = Refus
        NbRefus ++ ;
    sinon
        NbRefus -- ;
    fin si

ProcédureReceptionRequest :
    Horloge(i) = max(Horloge(i), heuredemande(j))
    Si demandeur(i) et (heuredemande(j) > heuredemande(i))
    alors differe(i)[j] = vrai; Envoyer < Refus, j >;
    fin si

ProcédureLibération :
    demandeur(i) = faux ;
    Pour j dans differe(i) faire
        Si differe(i)[j] alors
            Envoyer < Permission, j >
        fin si
    fin pour

```

10. On a fait jusqu'à maintenant l'hypothèse d'un maillage complet pour les communications c'est à dire un canal bidirectionnel entre tout couple de sites. La diffusion d'une requête peut être réalisée en lui faisant faire un tour d'anneau et l'envoi des permissions en les routant sur l'anneau vers leur destinataire. Le nombre de messages pour une utilisation de la section critique est alors égale à $\frac{n(n+1)}{2}$. Une meilleure solution (qui ne requiert que n messages=consiste, pour un site qui reçoit une requête, à ne la faire passer progresser que lorsqu'il donne sa permission : les messages de permissions deviennent alors inutiles et la reception par un site de sa propre requête, après un tour d'anneau, lui indique qu'il a reçu implicitement toutes les permissions.
11. Bien que les horloges puissent croître indéfiniment il n'en est pas de même de la différence entre les valeurs de deux horloges quelconques à un instant donné. En effet,

si h est la plus grande date associée à une requête, la prochaine date utilisée par une requête ne peut être supérieure à $h+1$. Comme il y a n sites et qu'un site ne peut faire une nouvelle requête que lorsque la précédente a été satisfaite, l'écart maximal entre les dates de deux requêtes est borné par $n-1$. Si l'on se place sur un site i on sait que donc que les autres horloges ont des valeurs dans l'intervalle $[h_i - (n-1), h_i + (n-1)]$ (les deux bornes de l'intervalle correspondent aux cas extrêmes où h_i serait respectivement la plus grande et la plus petite valeur d'horloge). Pour bien comprendre cet intervalle : soit h'_i l'horloge d'un site différent de i .

- Si h_i est la plus petite valeur des horloges alors $h_i \leq h'_i + (n-1)$, (ce cas apparaît quand les autres jouent entre-eux, et au dernier moment envoie le message à i).
- Si h_i est la plus grande valeur, il existe un site ayant la plus valeur d'horloge alors $h'_i - (n-1) \leq h_i$

Lorsqu'un site reçoit une requête estampillée, la date contenue dans celle-ci est donc nécessairement dans cet intervalle. On peut donc implémenter les horloges modulo p avec $p = 2n - 1$. $\log_2 p$ bits sont alors suffisants pour véhiculer les dates.

Fin correction exercice 1

Exercice 2 – L'algorithme de Carvalho et Roucairol Nous allons voir maintenant l'algorithme de Carvalho et Roucairol qui peut-être vu comme une amélioration du précédent. Considérons pour illustrer cela la situation suivante : deux sites, i et j tels que i veut utiliser la section critique plusieurs fois de suite alors que j n'est pas intéressé par celui-ci et reste donc dans l'état *dehors*. Avec l'algorithme précédent le site i demande la permission de j à chaque nouvelle invocation de l'opération acquérir. Le principe utilisé ici est le suivant : puisque j a donné sa permission à i , ce dernier la considère comme acquise jusqu'à ce que j demande à i sa permission ; dans la situation décrite le site i ne demande donc qu'une fois la permission à j .

1. Donner l'intervalle du nombre de messages nécessaires pour une utilisation de la section critique.
2. Donner une façon d'implémenter ce principe.
3. Montrer que la sûreté est garantie.
4. Montrer que la vivacité est garantie.
5. Donner l'algorithme.
6. Donner l'intervalle de temps de non utilisation de la section critique sachant que les messages ont un temps de transit borné par T .
7. Est-il possible de borner le domaine des valeurs d'horloges ?

8. Est-ce que le délai borné apporte quelque chose ?

Correction exercice 2

1. L'application de ce principe permet de réduire le nombre de messages nécessaires pour une utilisation de la section critique. Ce nombre est pair (à toutes requête correspond un renvoi de permission) et est fonction de l'état du système ; il varie entre 0 et $2(n-1)$.
2. Une façon d'implémenter l'idée précédente consiste à associer à tout couple de sites distincts i et j un et un seul message $permission(i, j)$ (dans ce qui suit $permission(i, j)$ et $permission(j, i)$ sont considérés comme deux synonymes du même message). Si ce message est chez i , celui-ci a la permission de j et inversement. Initialement $permission(i, j)$ est placé indifféremment sur l'un ou l'autre site. Lorsqu'un site i invoque l'opération $acquirir$ il doit demander les permissions qui lui manquent ; les sites qui détiennent ces permissions constituent donc l'ensemble R_i auquel i adresse ses requêtes :

$$R_i = \{j \text{ tel que le site } i \text{ ne possède pas } permission(i, j)\}$$

. Il faut donc maintenant gérer les ensemble R_i qui évoluent dans le temps. Lorsque le site i reçoit $permission(i, k)$ il supprime k de R_i ; lorsqu'il envoie $permission(i, m)$ il rajoute m à R_i . Si l'on considère deux sites quelconques i et j on doit donc avoir initialement (puisque le message $permission$ qui gère leurs conflits est chez un seul d'entre-eux).

$$i \in R_j \text{ ou exclusif } j \in R_i$$

. De plus les mises à jour ultérieures de R_i et R_j sont telles que cette relation est toujours vraie lorsque i et j invoque l'opération $acquirir$.

3. L'association à tout couple de sites d'une permission unique pour gérer leurs conflits garantit le sûreté.
4. L'association d'une estampille à toute requête va, comme pour l'algorithme précédent, garantir la vivacité. Lorsqu'un site i reçoit une requête estampillé (k, j) :
 - il renvoie $permission(i, j)$ s'il est dans l'état *dehors*
 - il mémorise le renvoi de $permission(i, j)$ s'il est dans l'état *dedans* ou s'il est prioritaire
 - enfin s'il est dans l'état *demandeur* et non prioritaire il renvoie $permission(i, j)$ au site j , suivi d'un message de requête pour indiquer au site j de lui retourner la permission après avoir utilisé la section critique.

Il est facile de montrer que la demande dotée de la plus petite estampille ne peut être indéfiniment bloquée.

5. l'algorithme :

Lors d'un appel à acquérir

```

etat_i = demandeur ;
last_i = h_i + 1 ;
∀ j ∈ R_i : envoyer requête(last_i, i) à j ;
attendre (R_i = ∅) ;
etat_i = dedans ;

```

Lors d'un appel à liberer :

```

etat_i = dehors ;
∀ j ∈ differe_i : envoyer permission(i, j) à j ;
R_i = differe_i ;
differe_i = ∅ ;

```

Lors de la reception de requete(k, j)

```

h_i = max(h_i, k) ;
priorité_i = (etat_i = dedans) ou ((etat_i = demandeur) et (last_i, i) < (k, j)) ;
Si priorité alors diffre_i = diffre_i ∪ {j} ;
Sinon ;
    envoyer permission(i, j) à j ;
    R_i = R_i ∪ {j} ;
    Si etat_i = demandeur ;
    Alors envoyer requete(last_i, i) à j ;
    fsi
fsi

```

Lors de la reception de permission(i, j) ;

```

R_i = R_i - {j} ;

```

6. $[0, 2T]$

7. A la différence de précédemment, il n'est pas possible de borner le domaine des valeurs d'horloges. En effet un site qui a donné toutes ses permissions et qui, après une longue période, émet une requête n'a pas incrémenté son horloge logique depuis la dernière requête reçue ; son horloge peut donc avoir une différence quelconque avec une autre

horloge. Ceci ne pouvait se produire avec le protocole précédent car la réception systématique de toute requête par tout site avait pour effet de bord de synchroniser les horloges locales modulo $n - 1$.

8. A priori oui

Fin correction exercice 2

Exercice 3 – Exclusion mutuelle sur un graphe complet Algorithme de Raymond

On suppose que l'on a un réseau logique complet à n sommets V . Pour tous les utilisateurs, une ressource à $M < n$ entrées est disponible. Nous proposons de définir un protocole d'accès à cette ressource. Voici une partie de ce protocole pour le sommet i . Les principales variables sont :

$permatt_i$: $array[1 \dots n]$ d'entiers positifs ;
 $differe_i$: $array[1 \dots n]$ d'entiers positifs ;
 $etat_i := dehors \in \{demandeur, dedans, dehors\}$;
 (* Les deux tableaux sont initialisés à 0 *)
 h_i est l'horloge associé au site i

Procédure Demander-SC

```

 $etat_i := demandeur$  ;
 $last_i := h_i + 1$  ;
 $nbperm_i := 0$  ;
Pour tout ( $j \in V - \{i\}$ ) faire
  Envoyer( $\langle REQUEST, last_i, i \rangle$ ) à  $j$  ;
   $permatt_i[j] := permatt_i[j] + 1$ 
Attendre( $nbperm_i \geq n - M$ ) ;
 $etat_i := dedans$  ;

```

< SECTION CRITIQUE >

(* Phase de libération *)

```

 $etat_i := dehors$  ;
Pour tout ( $j : j \neq i, differe_i[j] \neq 0$ ) faire
  Envoyer( $\langle PERMISSION, i, differe_i[j] \rangle$ ) à  $j$  ;
   $differe_i[j] := 0$  ;

```

Lors de la réception de $\langle REQUEST, k, j \rangle$

...

Lors de la réception de $\langle PERMISSION, j, x \rangle$;

```

 $permatt_i[j] := permatt_i[j] - x$  ;
Si ( $etat_i = demandeur$ ) alors
  Si ( $permatt_i[j] = 0$ ) alors  $nbperm_i := nbperm_i + 1$  ;

```

1. Si $M = 1$, que retrouve-t'on ?
2. Donner la signification de $permatt_u[v]$, $differe_u[v]$ et $nbperm_u$
3. Pourquoi dans la procédure Demander-SC attend on $n - M$ permissions avant d'entrer en section critique ? Que représente la donnée associée au message $\langle REQUEST, .. \rangle$? Comment déterminer des niveaux de priorité grâce à ces données ? Si un sommet u reçoit un message $\langle REQUEST, .. \rangle$ d'un sommet v plus prioritaire, et que lui-même demande une ressource, que doit-il faire ?
4. Donner un exemple à trois sites, pour lequel la procédure $\langle PERMISSION, j, x \rangle$ avec $x \geq 2$ est utilisée.
5. Ecrire la procédure de réception de $\langle REQUEST, .. \rangle$.
6. Montrez (par contradiction) qu'au plus M sommets peuvent simultanément utiliser la ressource.
7. Avec un tel algorithme, un site peut-il attendre infiniment avant d'entrer en section critique ?
8. Donner la complexité concernant le nombre de messages pour l'utilisation d'une ressource.

Correction exercice 3

1. Si $M = 1$ nous retrouvons l'algorithme de Ricart et Agrawala.
2. La case $permatt_u[v]$ est un compteur qui dénombre le nombre de permissions que u a demandé (via des messages $\langle REQUEST \rangle$). La case $differe_u[v]$ indique le nombre de demandes de permissions que u a reçu de v et que u n'a pas pu encore satisfaire (typiquement parce qu'il est en attente et que la demande de v est moins prioritaire). Lors de la phase de libération, u va « payer ses

dettes envers v » et lui renvoyer toutes les permissions en retard (voir algorithme). La variable $nbperm_u$ est un compteur qui compte le nombre de permissions reçues. C'est sur sa valeur que se fait l'entrée en section critique.

- Un site peut utiliser une entrée de la ressource dès qu'il sait qu'au plus $M-1$ demandes sont en cours de satisfaction ou plus prioritaires que la sienne. Le site u faisant $n-1$ demandes (voir algorithme) il doit donc attendre d'avoir reçu au moins $n-1-(M-1) = n-M$ permissions.

La données associée au message $\langle REQUEST, .. \rangle$ représente l'horloge logique du site qui envoie le message. On a la relation d'ordre suivante :

$$(h_i, i) < (h_j, j) \Leftrightarrow (h_i < h_j \text{ ou } (h_i = h_j \text{ et } i < j))$$

Ce qui peut se traduire par : i est plus prioritaire que j car :

soit la demande de i a été faite avant celle de j (en prenant en compte les horloges locales de i et j). Soit les horloges sont les mêmes mais l'identificateur de i est plus petit que celui de j .

NB : cet ordre est *total* (toutes les paires sont comparables). Un site moins prioritaire u doit «céder sa place» à un site plus prioritaire v , c'est-à-dire renvoyer une permission de u à v .

- Un site peut retarder le renvoi de plusieurs permissions vis à vis d'un autre site j . Considérons par exemple $n = 3$ sites, j, k et $M = 2$ ressources. Le site i obtient 2 permissions et utilise une des 2 ressources pendant une très longue période ; durant celle-ci le site j peut utiliser la seconde ressource plusieurs fois de suite grâce à la permission que lui renvoie le site k suite à chacune de ses requêtes. Le site i reçoit alors plusieurs requêtes de j et ne renverra ses permissions que lorsqu'il libéra sa ressources.
- Voici le code :

```
Lors de la réception de  $\langle REQUEST, k, j \rangle$  de  $j$ 
   $h_i := \max\{h_i, k\} + 1$ ;
   $priorite_i := (etat_i = dedans)$  ou  $((etat_i = demandeur)$  et  $(last_i, i) < (k, j)$ );
  Si  $(priorite_i)$  alors  $differe_i[j] := differe_i[j] + 1$ ;
  Sinon Envoyer  $\langle PERMISSION, i, 1 \rangle$  à  $j$ ;
```

Remarque : lorsqu'un site j reçoit un message $\langle REQUEST, h, i \rangle$ il recalc son horloge h_j par : $h_i := \max\{h_i, k\} + 1$ ceci garantit que toutes les requêtes futures du site j auront des dates supérieures à h .

- Supposons par contradiction qu'à un instant donné, $p > M$ sites aient accès à la

ressource. Les demandes sont datées et ordonnées comme suit :

$$(h_{i_1}, i_1) < (h_{i_2}, i_2) < \dots < (h_{i_M}, i_M) < (h_{i_{M+1}}, i_{M+1}) < \dots < (h_{i_p}, i_p) \quad (1)$$

Considérons le site i_{M+1} . Pour entrer en section critique, ce site a du recevoir des permissions d'au moins $n-M$ des $n-1$ sites. Donc, au plus $n-1-(n-M) = M-1$ sites n'ont pas envoyé de permission. Donc un site i_x avec $x \leq M$ a envoyé une permission au site i_{M+1} . Considérons la situation dans laquelle le site i_x a reçu la requête $(h_{i_{M+1}}, i_{M+1})$. Il y a trois cas à examiner :

- i_x était entrain d'exécuter une section critique ou attendait d'y rentrer avec une demande datée (h_{i_x}, i_x) . Dans ce cas, i_x n'aurait pas donné de permission.
- i_x n'était pas en section critique ni entrain d'y entrer. Dans ce cas, lors de la réception, l'horloge de i_x aurait pris une valeur supérieure à celle de $(h_{i_{M+1}}, i_{M+1})$. Ainsi, la future demande de i_x aurait une date plus récente que celle de i_{M+1} , ce qui contredit (1).
- Même cas que 1 mais avec une autre demande antérieure datée $(h, i_x) : (h, i_x) \leq (h_{i_x}, i_x)$. Dans ce cas aussi, lors de la réception, l'horloge de i_x aura pris une valeur supérieure à celle de $(h_{i_{M+1}}, i_{M+1})$. Ainsi, la demande de i_x aura une date supérieure à celle de i_{M+1} ce qui contredit (1).

Dans tous les cas il y a contradiction. Ainsi i_x n'a pas pu répondre à i_{M+1} qui ne peut donc pas être en section critique à l'instant t . Ainsi, il y a au plus M sites en section critique.

- Un site u peut-il attendre indéfiniment avant d'accéder à une ressource ? non car l'étiquetage est un ordre total. Lorsque tous ceux qui ont une étiquette plus petite seront sortis de la section critique, u pourra y entrer. En fait u pourra y rentrer lorsqu'un des $M-1$ derniers (avant lui) aura libéré sa place.
- L'utilisation d'une ressource critique engendre un nombre de messages compris entre $2(n-1)$ (si toute requête donne lieu à une permission) et $(2n-M-1)(n-1)$ requêtes provoquent au moins $M-n$ permissions). Le nombre moyen de messages dépend de l'état des demandes et des durées d'utilisation, un seul message $permission(j, x)$ remplaçant x messages $permission(j, x)$.

Fin correction exercice 3

Exercice 4 – Algorithme de Maekawa et ses variantes Dans l'algorithme de Ricart et Agrawala un site désirant la ressource doit demander la permission à tous les autres sites ce qui entraîne la génération d'un grand nombre de messages. Au lieu de demander si

grand nombre de permissions on peut imaginer que chaque site i a un ensemble de sites S_i à qui il va demander la permission d'entrer en section critique. Lorsque i obtient toutes les permissions de S_i il peut utiliser la ressource. Un tel ensemble S_i sera appelé *quorum*.

Nous souhaitons donc obtenir un algorithme réparti « symétrique » dans lequel chaque site joue un rôle équivalent au x autres au sens suivant :

$$(c_1) \quad \forall i : \text{Card}(R_i) = K$$

$$(c_2) \quad \forall i \quad i \text{ est contenu dans } D \text{ ensemble } R_j$$

1. Donner la signification des contraintes c_1 et c_2 .
2. Est-ce que l'algorithme de Ricart-Agrawala respecte les deux contraintes c_1 et c_2 . Si oui, avec quelles valeurs de K et de D . Si nous utilisons l'algorithme de Ricart-Agrawala avec les valeurs données ci-avant, quel est le nombre de messages nécessaires pour accéder à la section critique ? Conclusion ?
3. Donner la règle d'intersection, pour qu'on ait la propriété de sûreté.
4. On constate cependant qu'il peut y avoir des blocages si on ne met en place que ce mécanisme. Il faut rajouter un ordre de priorité pour s'assurer qu'en cas de multiples demandes simultanées au moins un site puisse recevoir toutes les permissions. Que va-t-on utiliser ?
5. Toutes ces propriétés sont indispensables pour assurer la sûreté et la vivacité. Cependant, d'autres critères peuvent être pris en compte. Parmi ceux-ci on peut distinguer la charge induite par le mécanisme pour chaque site ainsi que la performance en nombre de messages. Le premier point est relatif à la charge de travail que chaque site du système doit accomplir pour la communauté, même s'il n'a pas besoin de la ressource. Le deuxième point est la complexité de l'algorithme. Nous voudrions que chaque site i fasse partie de D quorums S et que pour tout j , $|S_j| = k$. Dans ce cas, la complexité de l'algorithme est en $O(k)$. Il faudrait donc minimiser k et D . Donner le nombre maximum de quorum qui peuvent être construits.
6. Construire effectivement des quorums avec ces propriétés est difficile (plans projectifs). Par contre, si $n = p^2$ il est facile de construire n quorums avec $\sqrt{n} = p$ sites dans chacun d'eux.
7. Dans ce qui suit nous décrivons les grandes lignes de l'algorithme d'allocation de ressources. Avant de mettre en place l'algorithme il faut au préalable construire les quorums et les affecter aux sites. Chaque site i connaît son quorum S_i . Chaque fois que i demande la ressource, il demande la permission, estampillée par son horloge, à tous

les sites de S_i . Lorsqu'un site i reçoit une demande de permission de la part de j on peut être dans un des trois cas.

- i ne demande pas la ressource. Il envoie sa permission à j .
- i a demandé la ressource. Si sa demande a une estampille plus ancienne que celle de j il lui envoie un message pour lui dire non, sinon il lui donne la permission.
- i a déjà donné sa permission à un site k dont l'heure de la demande avait une estampille plus récente que celle de j . Dans ce cas, i va essayer de récupérer la permission qu'il avait donné à k pour la donner à j . Si k a reçu un message de refus de permission, il redonne la permission à i qui va la donner à j .

Expliquer pourquoi il n'y a pas de blocage et que la vivacité est assurée.

8. Donner l'algorithme.
9. Donner un scénario pour lequel il existe un interblocage.
10. Comment résoudre le problème de l'interblocage ?
11. Dans quel cas il y a déséquencement ?
12. Donner le délai pour lequel la section critique bien que demandée n'est pas utilisée ?
13. Quel est le nombre de messages relatifs à une utilisation de la section critique ?

Correction exercice 4

1. La contrainte c_1 exprime que tous les sites doivent demander et obtenir le même nombre de permissions pour pouvoir entrer en section critique (règle du même effort) ; la contrainte c_2 indique que chaque site joue un rôle d'arbitre pour le même nombre de sites (règle de la même responsabilité).
2. Les r_i utilisés dans l'algorithme de Ricart-Agrawala, respectent les deux contraintes avec $K = D = n - 1$. Le nombre de messages nécessaires pour obtenir la section critique est de $3(n-1)$, c'est trop gourmand en nombre de messages. Donc cette classe d'algorithme est intéressante quand K et D sont raisonnables petits.
3. Pour être sûr que la propriété de sûreté est vérifiée il faut que les quorums vérifient la propriété suivante.

Règle d'intersection : si $i \neq j$ alors $S_i \cap S_j \neq \emptyset$

Cette propriété est indispensable pour s'assurer que lorsque deux sites i et j demandent les permissions pour entrer en section critique, les sites qui sont à la fois dans S_i et S_j ne peuvent pas accorder la permission aux deux. Ainsi, en cas de demandes simultanées au moins un des sites n'aura pas une permission.

4. Cet ordre sur les demandes sera celui des horloges logiques.
5. Avant tout il peut montrer que $D = K$ de la manière suivante : par la contraintes c_1 , en faisant la somme $\sum_i Card(R_i) = nk$. De plus, nous avons par la contraintes c_2 pour tous les sites nD . Ces deux quantités ne peuvent être égale.
Un quorum contient k membres qui chacun font partie d'au plus $D-1$ autres ensembles. Ainsi, le nombre maximum de quorums qui peuvent être construits est $n = k(D-1)+1$. De plus, $\frac{Dn}{k}$ est le nombre maximum de quorums car chaque site peut être dans au plus D quorums contenant chacun au plus k sites. S'il y a autant de sites que de quorums, on a $n = \frac{Dn}{k}$, c'est à dire $D = k$. En combinant ces égalités on obtient $n = k(k-1)+1$ et $k = O(\sqrt{n})$.
6. La figure 3 illustre le mécanisme pour le site x : les sites sont dessinés suivant une grille et, le quorum d'un site x est l'ensemble de tous les sites sur sa ligne et sur sa colonne dans la grille. Avec cette construction, toutes les bonnes propriétés sont vérifiées.

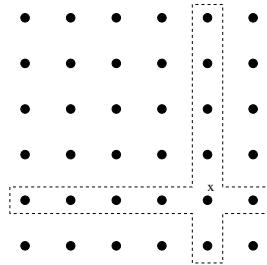


FIG. 3 – Illustration de la construction des quorums pour $n = 36$

7. Comme l'ordre basé sur les estampilles est un ordre total il y aura toujours un site qui pourra utiliser la ressource. Ceci garantit qu'il n'y a pas de blocage et que la vivacité est assurée. La sûreté provient du fait qu'un site ne donne sa permission qu'à au plus un site à chaque instant.
8. a faire
9. Examinons le scénario suivant : trois sites 1, 2 et 3 ont pour ensembles R_i respectivement $R_1 = \{1, 2\}$, $R_2 = \{2, 3\}$ et $R_3 = \{1, 3\}$, s'ils demandent simultanément la section critique l'interblocage est possible si le site 2 (resp. 3 et 1) donne l'unique permission qui gère à 1 (resp. 2, 3).
10. La résolution de l'interblocage est basée sur un mécanisme de préemption (mise en oeuvre par les messages *rendreperm*).

11. Pour résoudre ce problème il faut mettre en place un système de rendu de permissions.
12. Il y a déséquencement quand sur un site j_1 le message *rendreperm* arrive avant la permission.
13. Le délai durant lequel la section critique, bien que demandée n'est pas utilisée, est toujours égal à $2T$.
14. En conclusion, le nombre de messages relatifs à une utilisation de la section critique est compris entre $3 \times Card(R_i)$ (un message de requête + une permissions + une restitution de la permission) et $5 \times Card(R_i)$ (les 3 messages précédents plus les 2 messages éventuels nécessaires pour établir la priorité qui évitera les interblocages).

Fin correction exercice 4

Exercice 5 – Un algorithme mixte dû à Singhal

Dans les algorithme à permissions d'arbitres l'interblocage peut nécessiter, comme nous venons de la voir, la préemption de permissions. Peut-on dans cette classe d'algorithmes, une permission une fois attribuée, ne jamais la retirer?

1. Proposer une idée en vous basant sur les algorithmes précédents.
2. Appelons par CR_i l'ensemble $CR_i = \{j \text{ tel que } i \in R_j\}$. Quel est la signification? Peut-on avoir $R_i \cap CR_i \neq \emptyset$?
3. Voici une partir de l'algorithme. Compléter les parties manquantes (parties *retourperm* et *reque*) et expliquer le fonctionnement de l'algorithme.

Lors d'un appel à acquérir

```

etati = demandeur ;
hi = hi + 1 ; lasti := hi ;
attendusi := Ri ;
∀ j ∈ Ri : envoyer requête(lasti, i) à j ;
insérer(lasti, i) dans filei (la file est triée α)
attendre (Ri = ∅) et ((lasti, i) est en tête de filei ; (β)
etati = dedans ;

```

Lors d'un appel à liberer :

```

supprimer (lasti, i) de filei
etati = dehors ;
∀ j ∈ Ri : envoyer retourpermission(hi, i) à j ;
∀ m tel que (x, m) ∈ filei : envoyer permission(hi, i) à m (γ)

```

Lors de la reception de $permission(k, j)$;
 $h_i := \max(h_i, k)$
 $attendus_i := attendus_i - \{j\}$

4. Quel est le rôle de $file_i$?
5. Comment pouvez-vous définir les ensembles R_i ?
6. Quel est le délai de non utilisation de section critique ?

Correction exercice 5

1. Pour répondre à cette question considérons la permission que gère un site i comme un verrou ; avec le schéma précédent il s'agit de verrou de type exclusif. S'interdire de retirer un verrou (permission) pour le donner à un autre, conduit à utiliser un verrou de type partagé ; en d'autres termes, à un instant donné, suite à plusieurs requêtes un site peut avoir donné plusieurs fois sa permission. Avec ce schéma d'attribution des permissions la contrainte initiale : $\forall i, j R_i \cap R_j \neq \emptyset$ n'est pas assez forte pour garantir la sûreté, chaque site appartenant à cette intersection pouvant avoir donné sa permission à i et à j . L'utilisation de l'estampillage qui crée un ordre sur les requêtes, rend suffisant pour tout couple de sites la contraintes suivante :

$$\forall i \neq j i \in R_j \text{ ou } j \in R_i$$

Cette contrainte assure en effet que , pour tout couple de sites, au moins l'un est arbitre pour l'autre et donc que tout couple peut gérer ses conflits ; de plus comme l'estampillage est unique, tous les conflits seront résolus dans le même sens.

On peut constater que d'un part cette contrainte est plus faible que celle de l'algorithme de Ricart-Agrawala (un *ou* remplace ici le *et*) et que d'autre part elle ne se distingue de celle de Carvalho et Roucairol que par le remplacement du *ou exclusif* par un *ou*. Ce sont ces constatations qui donnent le qualificatif de "mixte" à cet algorithme.

2. Il s'agit des sites qui demandent la permissions à i ; on a de plus $R_i \cup CR_i \cup \{i\} = \{1, 2, \dots, n\}$ c'est à dire tous les sites (la contrainte indiquée n'interdit pas d'avoir $R_i \cap CR_i \neq \emptyset$).
3. Tout site i gère une file d'attente $file_i$ dans laquelle il place les estampilles des requêtes qu'il reçoit (issues des sites de CR_i) et celle de sa propre requête s'il a fait une demande demande ; cette file est triée dans l'ordre croissant des estampilles. Le site i peut entrer en section critique dès qu'il a reçu toutes les permissions attendues des sites de R_i (il

est alors prioritaire par rapport à eux) et que l'estampille de sa requête est en tête de $file_i$ (il est alors prioritaire par rapport aux sites de CR_i). Lorsqu'un site sort de section critique il renvoie les permissions pour chaque requête qui est dans la $file_i$ (c'est à dire aux sites de CR_i qui lui en ont demandé) et retourne celles qu'il avait obtenues aux sites de R_i (à l'aide de messages *retourperm*).

Afin de garantir une progression correcte des horloges, et donc la vivacité, tous les messages sont estampillés. Les autres variables utilisées ont la signification habituelle.

Lors de la reception de $requete(k, j)$

$$h_i = \max(h_i, k) ;$$

priorité $_i = (etat_i = dedans)$ ou $((etat_i = demandeur)$ et $(last_i, i) < (k, j))$;

Si non **priorité** alors envoyer $permission(h_i, j)$ à j ;

insérer (k, j) dans $file_i$ (δ)

Lors de la reception de $retourmperm(k, j)$;

$$h_i := \max(h_i, k)$$

supprimer (k, j) de $file_i$ (ϵ)

4. Le rôle de $file_i$ utilisée joue en fait deux rôles. L'un est semblable à celui que joue l'ensemble $diffri$ dans l'algorithme de Ricart et Agrawala : elle mémorise les sites auxquels il faudra renvoyer des permissions à la sortie de section critique : ligne γ . Le second rôle est l'établissement de la priorité de i par rapport aux sites de CR_i qui ont fait des requêtes reçues par i (lignes α , β , δ , ϵ) ; cette propriété est fondée sur les estampilles (ligne β) ; de plus l'estampille de la requête d'un site $j \in CR_i$ est placée dans la file à la ligne δ , elle en est supprimée à la ligne ϵ .
5. Pour être optimaux les ensembles R_i (définis statiquement) doivent être les plus petits possibles (c'est alors que globalement on aura le moins de messages échangés). ceci revient à remplacer le *ou* de la condition par le *ou exclusif*, c'est à dire $\forall i \neq j : i \in R_j$ ou exclusif $j \in R_i$. Plusieurs possibilité existent donc pour définir des R_i optimaux. On peut par exemple avoir la définition en escalier :
 $\forall i \in 2 \dots n : R_i = \{i - 1, i - 2, \dots, 1\}$ et $R_1 = \emptyset$.
 Une définition plus équilibré est possible, chaque ensemble R_i ayant $n/2$ éléments. Dans ce cas, avec des R_i optimaux, le nombre moyen de messages par utilisation de la section critique est égal à $3(n - 1)/2$. Cet algorithme est donc, avec des R_i optimaux, meilleur que celui de Ricart et Agrawala qui requiert $2(n - 1)$ messages.
6. Le délai durant lequel la section critique, bien que demandée, est inutilisée est 0, T ou

$2T$ avec la définition en escalier, et T ou $2T$ avec la définition équilibré.

Fin correction exercice 5

Exercice 6 – L’algorithme de Naimi et Tréhel Dans cet exercice nous nous plaçons dans le cas d’une ressource à une seule entrée (au plus un sommet peut utiliser la ressource à un instant donné). Réaliser l’exclusion mutuelle en réparti revient à proposer une mise en oeuvre distribué de l’objet informatique qu’est une file des sites. plusieurs solutions sont envisageables allant de la duplication de l’objet sur chaque site (avec un protocole qui assure la cohérence mutuelle des copies) à son partitionnement ; L’algorithme que nous présentons maintenant appartient à cette dernière catégorie. Une file va donc être implémentée en plaçant sur chaque site i un pointeur $suivant_i$, qui, s’il est différent de nil , indique le site successeur de i dans la file d’attente. La file étant définie deux problèmes se posent :

1. Comment un site sait-il qu’il est en tête de la file ?
2. Comment un site qui n’est pas dans la file peut-il venir se placer en queue de file ?

C’est la réponse à ces deux questions qui va définir l’algorithme.

1. Proposez un système qui permet de savoir si un site est en tête de file ?
2. Proposez une structure qui permet de se reconfigurer dynamiquement et qui permet de savoir qui se trouve en dernier (dernier élément de la file).
3. Proposez un protocole de gestion de la structure
4. Nous avons raisonné jusqu’à maintenant comme si la file n’est pas vide. Proposez une solution simple pour éviter que la file soit vide.
5. Donner un exemple d’utilisation de l’algorithme où s est la racine de l’arbre possédant le jeton et p et q désirent entrer dans la file. p est traité avant q et p reçoit le jeton de s . Puis s veut rentrer en section critique.
6. Ecrire la phase de libération.
7. Donner un exemple d’exécution de cet algorithme. A qui correspondent les variables $pere_p$ et $suivant_p$?
8. Pourquoi un sommet demandant le jeton l’obtiendra t’il au bout d’un temps fini (s’il n’y a pas de panne) ?
9. Quel est le nombre de messages échangés dans le pire des cas lors d’une demande ?
10. Est-ce que pour un site i son père diffère durant l’exécution ? Qu’est-ce que cela implique pour tous les sites ?

Voici le protocole que nous proposons. Le code est décrit pour le sommet p .

```
Procédure Demander-SC ;
   $demandeur_p := vrai$  ;
  Si ( $pere_i \neq nil$ ) alors
    Envoyer( $\langle REQUEST \rangle$ ) à  $pere_p$  ;
     $pere_p := nil$  ;
    Attendre( $AvoirJeton_p$ ) ;
```

<SECTION CRITIQUE>

(* Phase de libération *)

```
Lors de la réception de  $\langle REQUEST \rangle$  de  $q$  ( $q$  demandeur) ;
  ( $Demandeur_p$ ) alors
    ( $Suivant_p = nil$ ) alors  $suivant_p := q$ 
  Si ( $pere_p \neq nil$ ) alors
    Envoyer( $\langle REQUEST \rangle$ ) à  $pere_p$ 
  Sinon Si ( $AvoirJeton_p$ ) alors
     $AvoirJeton_p := faux$ 
    Envoyer( $\langle TOKEN \rangle$ ) à  $q$ 
     $pere_p := q$ 
```

```
Procédure recevoir  $\langle TOKEN \rangle$  de  $q$  ;
   $AvoirJeton_p := vrai$  ;
```

Initialisation : choisir un sommet S quelconque et

```
 $AvoirJeton_S := vrai$  ; Pour tout  $X \neq S$ ,  $AvoirJeton_X := faux$  ;
Pour tout sommet  $Y$ ,  $pere_Y := S$  ;  $suivant_Y := NIL$  ;  $Demandeur_Y := faux$  ;
```

Correction exercice 6

1. La tête de file est unique et seul le site qui est en tête de file a besoin de la savoir ; il peut indiquer à son successeur que ce dernier devient la nouvelle tête de file lorsque lui-même en sort. Les propriétés que nous venons de décrire peuvent trivialement être

mises en oeuvre par un jeton : être en section critique c'est l'avoir, et sortir de la file revient à passer le jeton à son successeur.

2. Le dernier élément de la file a, tout comme la tête, une propriété d'unicité, mais contrairement à celle-ci, le fait de ne plus être le dernier de la file ne dépend pas du dernier : tout site peut faire une requête et devenir alors le dernier. Il est donc nécessaire que tout site puisse adresser le dernier site de la file pour lui indiquer qu'il n'est plus le dernier et qu'il a désormais un successeur. En d'autres termes il faut fournir aux sites une structure d'adressage qui d'une part leur permette de savoir qui est le dernier et qui d'autre part puisse se reconfigurer dynamiquement lorsqu'un site rentre dans la file devenant de ce fait le "nouveau dernier". Une arborescente couvrante de racine ce dernier est une telle structure (le dernier de la file est le sommet racine de l'arborescence) :
 - elle présente une propriété d'unicité (la racine est unique : il s'agit du dernier élément de la file).
 - elle offre un adressage qui permet à tout site d'adresser (indirectement) la racine.
 - elle peut facilement se reconfigurer en inversant le sens des arcs ou en détruisant certains arcs et en rajoutant d'autres. (cette technique est connue sous le nom d'inversion d'arcs "edges or path reversal").
3. Le protocole de gestion de cette structure est alors le suivant. Lorsqu'un site i veut entrer dans la file il envoie un message *requete*(i) à son père dans l'arborescence puis ce déclare racine, il n'a plus alors de père. Lorsqu'un site k reçoit un message *requete*(i) il le fait progresser vers son père (pour qu'il atteigne la racine) puis déclare que son nouveau père est le site i : à la connaissance de k c'est le site i qui est maintenant le dernier de la file. Lorsqu'un site m qui est racine reçoit un message *requete*(i) il apprend qu'il n'est plus le dernier et, outre la mise à jour *suivant* $_m = i$, il déclare que son père est le site i . L'arborescence couvrante est ainsi reconfigurée. Il est important de noter de remarquer que seule la branche comprise entre la nouvelle et l'ancienne racine est modifiée par ce protocole de reconfiguration. De plus, tout site doit appartenir à l'arborescence. En effet, lorsqu'un site sort de la file l'arborescence n'est pas modifiée ; or s'il veut rentrer à nouveau dans la file il doit être dans l'arborescence afin d'en connaître le dernier élément. Le jeton qui gère la tête de la file et l'arborescence qui en gère le dernier élément sont ainsi deux "structures de contrôles" utilisées de façon orthogonale (au sens où elles ne sont pas réductibles l'une à l'autre).
4. Le cas de la file vide : nous avons raisonné jusqu'à présent comme si la file n'était jamais vide. Qu'en est-il lorsque cela se produit ? Pour cela examinons le cas où il n'y a qu'un seul site dans la file, soit i ce site. Il est à la fois le premier et le dernier élément de la file et donc possède le jeton (*jetonpresent* $_i$ est à *vrai*) et est la racine

de l'arborescence. Lorsqu'il sort de la file il ne peut transmettre le jeton puisque la file est alors vide : il reste donc possesseur du jeton et racine. Afin de distinguer les deux possibilités précédentes il suffit d'introduire un booléen local à chaque site *danslafile* $_i$ qui a la valeur *vrai* si, et seulement si, le site i est dans la file. On a alors

$$\exists i(\text{non } \textit{danslafile}_i \text{ et } (i \text{ est racine ou } \textit{jetonpresent}_i))$$

est équivalent à *la file est vide*.

5. L'exécution est donné par la figure ??.
6.

```
Si (Suivant $_p \neq \textit{NIL}$ ) alors
  AvoirJeton $_p := \textit{faux}$  ;
  Envoyer(< TOKEN >) à Suivant $_p$  ;
  Suivant $_p := \textit{NIL}$  ;
  Demandeur $_p := \textit{faux}$ 
```
7. On prend un exemple à 5 sites. Voir le déroulement à la figure4. Les arcs pleins pointent vers le prochain site à qui on va demander le jeton la prochaine fois (même si ce n'est pas lui qui l'a). La racine de l'anti arborescence est le sommet qui a ou aura le jeton (la variable *NowelleRacine* représente l'adresse de cette racine). Les arcs en pointillés représentent une **file d'attente répartie**. Lorsqu'un site demande le jeton il va être placé à la fin de cette file. Lorsque le site qui a le jeton le libère, il le passe à son successeur dans la file (variable *Suivant*).
8. Avec toutes les hypothèses, le site possédant le jeton va finir par le faire passer au suivant dans la file. Or, il est toujours possible de s'insérer dans la file. Ainsi, l'attente du jeton est finie.
9. Nombre de messages dans le pire des cas : le message < *REQUEST* > peut être envoyé jusqu'à au plus $n - 1$ fois et le jeton est envoyé une fois. Ainsi, au plus n messages échangés lors d'une demande de jeton. On montre (dur) qu'en moyenne il y a $O(\log n)$ messages échangés.
10. L'algorithme étudié est tributaire de la connaissance des identités des sites : lorsqu'un site i fait suivre *REQUEST*(j), l'identité j a une signification dont la portée est globale.
11. Le nombre de messages nécessaires à une utilisation de la section critique est compris entre 0 et n .

12. La valeur n est obtenue lorsque l'arborescence étant dégénérée, le site i (la seule feuille) requiert la section critique alors que le jeton est possédé par le site m (le père), $n - 1$ messages de requête vont être utilisés plus un message pour le jeton.
13. Remarquons simplement que l'arborescence qui fait suite à une arborescence dégénérée est optimale c'est à dire $n - 1$ feuilles.
14. Le délai durant lequel la section critique, bien que demandée est inutilisé varie entre 0 et nT .

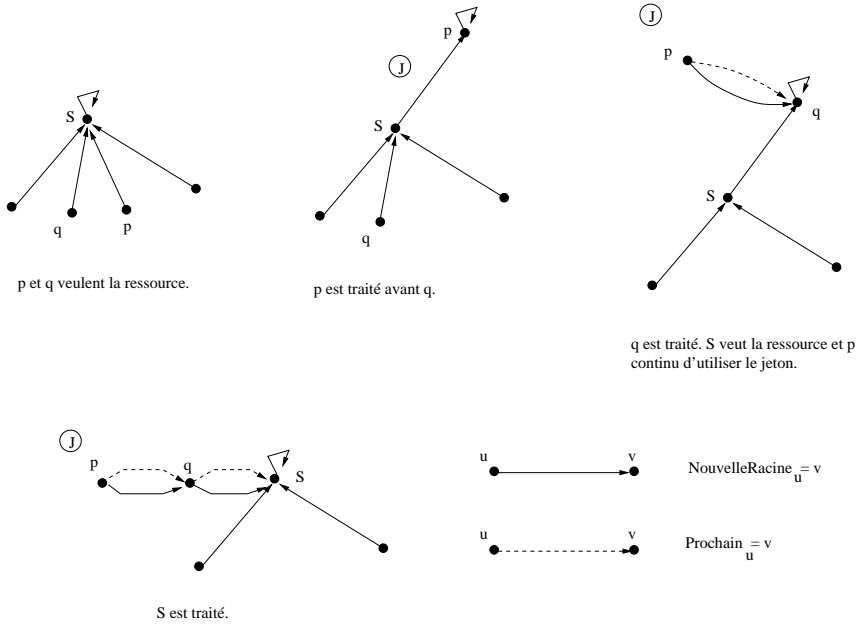


FIG. 4 – Un exemple d'exécution

Fin correction exercice 6

Algorithme distribué
TD – Séance n° 3

Exercice 1 – Construction d’une arborescence des plus courts chemins en largeur

Nous souhaitons construire une arborescence

1. Construire une arborescence en partant d’un initiateur noté P_α . Donner la complexité de l’algorithme. Qui détecte la terminaison?
2. Nous souhaitons construire une arborescence des plus court chemins.
 - (a) En se basant sur la question précédente, donner un algorithme qui construit une arborescence des plus court chemins.
 - (b) Evaluer la complexité dans le cas d’un graphe complet.
 - (c) Qui détecte la terminaison?
 - (d) donner une trace dans le cas d’un graphe compelt à 5 sommets avec P_1 m’initiateur. Donner l’arborescence construite.
3. Construire une arborescence des plus court chemins en largeur. Evaluer la complexité
4. Comment étendre ces résultats dans le cas d’un arbre?

Correction exercice 1

1. Pour l’arborescence , voici le code :

Le site P_α :

```
atteint $_\alpha$  := vrai ; pere $_\alpha$  :=  $\alpha$  ; fils $_\alpha$  :=  $\emptyset$       Nbatt $_\alpha$  := card(voisin $_\alpha$ ) ;  
 $\forall k \in$  (voisin $_\alpha$ ) Envoyer(explorer)  $P_k$  ;  
attendre Nbatt $_\alpha$  = 0
```

A la réception de *retour(b)* depuis P_j

$Nbatt_\alpha := Nbatt_\alpha - 1$

Si $b = oui$ alors $fils_\alpha := fils_\alpha \cup \{j\}$;

A la réception de *explorer* depuis P_j

Si *atteint $_i$* alors

envoyer *retour(non)* à P_j

Sinon $atteint_i := vrai$; $pere_i := j$; $fils_i := \emptyset$

$Nbatt_\alpha := card(voisin_i - \{j\})$;

$\forall k \in$ (voisin $_i - \{j\}$) Envoyer(explorer) P_k ;

attendre $Nbatt_i = 0$

renvoyer *retour(oui)* à $pere_i$

A la réception de *retour(b)* depuis P_j

$Nbatt_\alpha := Nbatt_\alpha - 1$

Si $b = oui$ alors $fils_\alpha := fils_\alpha \cup \{j\}$;

La complexité

Tout processus P_i , différent de l’initiateur, émet : $delta_i - 1$ messages *explorer* (où δ_i est son degré) et l’initiateur en émet δ_i . Il ya atant de message *retour* que de message *explorer* ; la complexité en nombre de messages est donc $O(e)$, avec e le nombre de canaux. La complexité temporelle est $O(n)$, où n est le nombre de processus, puis c’est la hauteur maximum de l’arborescence construite.

2. Pour l’arborescence des plus court chemins voici le code :

- (a) Il est important de noter que ce n’est pas en largeur que je construits. Il est donc nécessaire de tester si $r + 1 < niveau_i$

Le site P_α :

$atteint_\alpha := vrai$; $pere_\alpha := \alpha$; $fils_\alpha := \emptyset$

$Nbatt_\alpha := card(voisin_\alpha)$;

$\forall k \in$ (voisin $_\alpha$) Envoyer(explorer(0)) P_k ;

attendre $Nbatt_\alpha = 0$

A la réception de *retour(b,r)* depuis P_j

$Nbatt_\alpha := Nbatt_\alpha - 1$

Si $b = oui$ alors $fils_\alpha := fils_\alpha \cup \{j\}$;

A la réception de *explorer(r)* depuis P_j

Si *non(ateint $_i$)* alors

$atteint_i := vrai$; $pere_i := j$; $fils_i := \emptyset$; $niveau_i := r + 1$;

$Nbatt_\alpha := card(voisin_i - \{j\})$;

```

     $\forall k \in (\text{voisin}_i - \{j\})$  Envoyer(explorer(niveaui))  $P_k$  ;
    attendre  $Nbatt_i = 0$ 
    renvoyer retour(oui, niveaui) à perei
Sinon
    Si  $r + 1 < \text{niveau}_i$ 
    envoyer retour(non, niveaui) à  $P_{pere}$ 
     $pere_i := j$  ;  $niveau_i := r + 1$  ;  $fil_s_i := \emptyset$  ;       $Nbatt_\alpha := \text{card}(\text{voisin}_i -$ 
     $\{j\})$  ;
     $\forall k \in (\text{voisin}_i - \{j\})$  Envoyer(explorer(niveaui))  $P_k$  ;
    attendre  $Nbatt_i = 0$ 
    envoyer retour(oui, niveaui) à perei
    Sinon
        envoyer retour(non, r + 1) à  $P_j$ 

```

A la réception de *retour*(*b, r*) depuis P_j

```

Si  $r - 1 = \text{niveau}_i$  alors
     $Nbatt_\alpha := Nbatt_\alpha - 1$ 
    Si  $b = \text{oui}$  alors  $fil_s_\alpha := fil_s_\alpha \cup \{j\}$  ;
    Sinon
         $fil_s_\alpha := fil_s_\alpha - \{j\}$ 

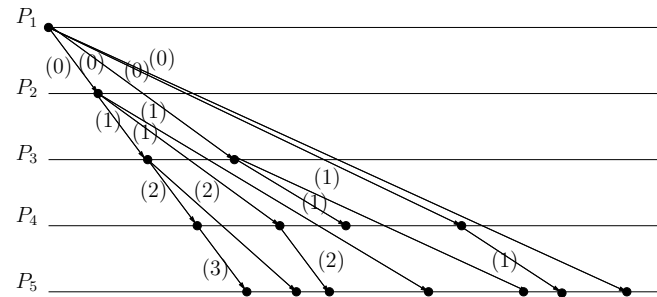
```

- (b) Le pire des cas est donné lorsque le réseau est un graphe complet à n sommets, et où le niveau de chaque processus est modifié un nombre maximum de fois. Parmi les processus différents de P_α , un est atteint une fois, un est atteint deux fois, ..., un est atteint $n - 1$ fois. Par atteint, on entend ici la réception d'un message *explorer* qui donne une nouvelle valeur *niveau_i*. Chaque fois qu'un site est atteint, il prolonge la vague vers ses $n - 2$ voisins (tous moins son père et lui-même). Le nombre total de messages explorer est donc :

$$(n - 1) + (n - 2) \sum_{i=1}^{n-1} i = \frac{(n - 1)(n^2 - 2n + 2)}{2}$$

Il y a autant de message retour, soit au total $O(n^3)$. La complexité temporelle est en $O(n)$.

- (c) Celui qui détecte la terminaison c'est le sommet α .
- (d) LA solution est donnée par la figure 1. L'arborescence est enracinée en P_1 et de hauteur un.



P_1 est l'initiateur

Seuls les messages *explorer* de P_i à P_j avec $i < j$ sont représentés

Un point sur la ligne d'un processus correspond au prolongement d'une vague

FIG. 1 – Exemple de trace pour la construction d'un arborescence des plus court chemins

3. Nous pouvons passer de la construction d'une arborescence à la construction d'un arbre.

Le problème consiste à construire une arborescence de recouvrement de ce réseau, dont la racine est un processus donné P_α . Cette arborescence doit être construite en largeur d'abord : si un processus P_i se trouve, dans le réseau, à distance k la racine, alors le chemin de P_α à P_i dans l'arborescence doit être de longueur k ; de plus, le processus P_i doit savoir qu'il est au niveau k , et doit connaître lequel de ses voisins constitue son père dans l'arborescence : cette dernière information permet un routage ascendant vers la racine, optimum par rapport au nombre de canaux utilisés ; il devra aussi connaître lesquels de ses voisins constituent ses fils dans l'arborescence.

Outre le fait d'exhiber un mécanisme de synchronisation cet algorithme est intéressant par son résultat. Une fois l'arborescence construite, elle fournit une structure de parcours du réseau des processus, qui peut être mise à profit comme constituant élémentaire d'algorithmes distribués plus sophistiqués ; par exempl, la diffusion d'une information depuis P_α vers tous les sites est triviale si l'on dispose d'une telle structure ; elle est de plus optimale car elle ne requiert que $n - 1$ messages (où n est le nombre de sites du réseau). Comme le passage d'une arborescence à un arbre (dans lequel il n'y a pas de racine et où les liaisons ne sont pas orientées) est immédiat, cette structure permet aussi la diffusion d'un message depuis n'importe quel processus vers chacun des autres, en exactement $n - 1$ messages.

Principe : le processus racine P_α va jouer un rôle privilégié. Il est le seul actif initialement et envoie à chacun de ses voisins un message *aller* ; il attend ensuite d'avoir reçu de chacun d'eux un message *retour*. Du point de vue de P_α , ceci constitue une phase de calcul. L'algorithme va consister en une succession de phases de calcul. L'algorithme va consister en une succession de phases synchronisées par P_α qui les séquencer l'une à la suite de l'autre. Il s'agit, du point de vue de P_α , d'un calcul itératif (distribué) : la progression des messages *aller* lors d'un pas de l'itération constitue une *vague* réfléchie par les messages *retour*. Le premier pas de l'itération va construire l'arborescence des plus courts chemins de profondeur 1, le second celle des plus courts chemins de profondeur 2, et ainsi de suite, chaque nouvelle arborescence étant le prolongement de l'arborescence construite par la vague précédente (en ce sens, il s'agit d'un algorithme glouton : le travail effectué jusqu'à la vague p est exploité, mais non remis en cause, par la vague $p + 1$).

La mise en oeuvre de ce principe est relativement simple. Elle consiste essentiellement à maintenir invariante la propriété suivante : (posons d_i = distance de P_α à P_i dans le réseau).

- pour $p \geq 1$ et tout processus P_i :
 - $d_i > p$, P_i n'est pas atteint par la vague p
 - $d_i = p$ P_i va apprendre qu'il est dans l'arborescence lors de la phase aller de la vague p par la même occasion, il apprend l'identité de son père et sa distance à P_α (profondeur). Ces informations sont stables : elles ne seront jamais remises en question.
 - $d_i < p$ P_i connaît de plus l'ensemble de ses fils dans l'arborescence construite ; cette information a été acquise lors du retour de la vague $d + 1$ (ou lors de la vague d_i si P_i a un seul voisin dans le réseau ; son père) ; une fois acquise elle est également stable.

Lorsqu'un processus P_i est dans l'arborescence il doit de plus coopérer à la construction pour les processus P_j tels que $d_j > d_i$; pour cela il doit être capable de relayer les vagues de numéro supérieur à d_i vers ceux de ses fils qui sont susceptibles de prolonger l'arborescence courante. Cette tâche est effectuée par P_i en maintenant un ensemble lui indiquant vers lesquels de ses fils il y a lieu de relayer les vagues successives.

Comment l'invariant précédent peut-être maintenu ? Afin d'apprendre sa distance à P_α et l'identité de son père (lors de la vague numéro d_i), les messages *aller* qui font progresser la vague vont transporter la distance de leur émetteur à P_α . Il est clair que cette information est facilement maintenue par les émetteurs qui connaissent déjà leur distance à P_α , c'est à dire ceux qui ont appris lors d'une vague précédente leur appartenance à l'arborescence. D'autre part, un processus P_i qui reçoit pour la première

fois un message *aller*, apprend qu'il entre dans l'arborescence, mémorise l'émetteur de ce message comme étant son père, et sa distance comme celle de son père augmentée de un ; il réfléchit ensuite la vague à l'aide d'un message retour dans lequel il place l'information suivante : *continuer* s'il a d'autres voisins que son père, *termine* sinon ; cette information permettra au père, lorsqu'il la recevra, de savoir s'il doit ou non maintenir son fils P_i dans l'ensemble de ceux qui peuvent prolonger l'arborescence lors des vagues ultérieures.

Lorsque P_i reçoit un message *aller* et se trouve déjà dans l'arborescence, cela indique qu'il est sollicité pour prolonger l'arborescence. Deux cas peuvent se présenter ; si l'émetteur P_j de ce message n'est pas le père de P_i , ce dernier réfléchit immédiatement la vague par un message *retour* dans lequel il place l'information *deja marque* pour lui indiquer qu'il a déjà un père. Au reçu de cette information, P_j rayera P_i de l'ensemble de ses fils et ne communiquera plus avec lui. Si par contre P_j est le père de P_i , ce dernier relaie le message *aller* vers ses voisins à l'exception de P_j et de ceux qui lui ont déjà envoyé des messages *retour(termine)* ou *retour(deja marque)* ; il attend alors de recevoir un message *retour* de chacun des voisins à qui il a envoyé *aller*, avant d'envoyer lui-même à son père un message *retour* porteur de l'information adéquate *continuer*-dans le cas où au moins un de ses fils lui a envoyé *retour(continuer)*- ou *termine* dans le cas contraire. L'algorithme est terminé lorsque tous les messages *retour* reçus par P_α à la fin d'une vague portent l'information *termine*.

- *aller(k)* où k est un entier est égal à la profondeur de l'émetteur augmentée de un.
- *retour(rep)* $rep \in \{\text{continuer, termine, deja marque}\}$
- Le contexte local d'un processus est le suivant :
 - *const voisins_i* = { identités des voisins de P_i }
 - *var*
 - *marque_i* : booléen init *faux* (appartenance de P_i à l'arborescence)
 - *pere_i* : identité
 - *fils_i* ensemble d'identité (position de P_i dans l'arborescence)
 - *profondeur_i* : naturel
 - *encom_i* ensemble d'identités (ceux des fils de P_i vers lesquels les messages *aller* doivent être relays ; a envoyé *aller* à P_j)
 - *repondu_i* : tableau[voisins_i] de booléen (*repondu_i[j]* = *faux* ssi P_i et n'a pas reçu le *retour* correspondant)

Le processus P_α est la racine de l'arbre et c'est le seul à avoir *pere_i = i*.

A la réception de < init >

marque_i := vrai ;

pere_i := i ;

$fil_s_i := \emptyset$
 $profondeur_i := 0$
 $encom_i := voisins_i$
 Si $encom_i = \emptyset$ alors algorithme termine pour P_i
 Sinon $\forall l \in encom_i$
 envoyer $aller(profondeur_i + 1)$ à P_l
 $repondu_i[l] := faux$

A la réception de $aller(k)$ depuis P_j

Si non $marque_i$ alors
 $marque_i := vrai$;
 $pere_i := j$
 $fil_s_i := \emptyset$
 $profondeur_i := k$
 $encom_i := voisins_i - \{j\}$
 Si $encom_i = \emptyset$ alors
 envoyer $retour(termine)$ à P_j
 Sinon envoyer $retour(continuer)$ à P_j
 Sinon Si $pere_i = j$
 $\forall l \in encom_i$
 envoyer $aller(profondeur_i + 1)$ à P_l
 $repondu_i[l] := faux$
 Sinon envoyer $retour(deja\ marque_i)$ à P_j

A la réception de $retour(x)$ depuis P_j

$repondu_i[l] := vrai$
 Si $x = continuer$ alors $fil_s_i := fil_s_i \cup \{j\}$
 Sinon Si $x = deja\ marque$ alors $encom_i := encom_i - \{j\}$
 Sinon si $x = termine$ alors $fil_s_i := fil_s_i \cup \{j\}$; $encom_i := encom_i - \{j\}$
 Si $encom_i = \emptyset$ alors Si $pere_i = i$ alors Arborescence construite
 Sinon envoyer $retour(termine)$ à P_{pere_i}
 Sinon Si $\forall j \in encom_i$; $repondu_i[j]$ alors
 Si $pere_i = i$ alors $\forall l \in encom_i$; envoyer $aller(profondeur_i + 1)$ à P_l ; $repondu_i[l] := faux$
 Sinon envoyer $retour(continuer)$ à $pere_i$

4. La complexité de cet algorithme est en $O(n^2)$. Pour cela, nous considérons le pire

des cas c'est le cas d'une chaîne. Les complexités en nombre de messages et en temps (si l'on suppose que le temps de transit et de traitement associé à chaque message est d'une unité de temps) sont $O(n^2)$ où n est le nombre de processus. (pour calculer ces complexités, on se place dans le pire des cas qui est atteint lorsque le réseau initial est une chaîne). $\sum_{i=1}^n (n-i) = O(n^2)$ pour les messages du type continuer et déjà marqué. La complexité de messages du type terminé est en $O(m)$.

5. La construction d'un arbre recouvrant peut se déduire immédiatement de celle d'une arborescence. Un arbre se déduit en effet d'une arborescence par la suppression de l'orientation des arcs. Les attributs $pere_i$ et fil_s_i de chaque processus P_i sont réunis en un seul ensemble : vda_i définissant les voisins de P_i dans l'arbre. On ne peut plus alors distinguer un unique sommet particulier qui n'aurait pas de père (la racine) ; par contre, les feuilles de l'arborescence (caractérisées par la condition $fil_s_i = \emptyset$) deviennent les sommets pendants de l'arbre ; ils sont caractérisés par $card(vda_i) = 1$.
 Nous ajoutons : $vda_i := fil_s_i \cup \{pere_i\}$ (cas où $pere_i \neq i$) et $vda_i := fil_s_i$ sinon.

Fin correction exercice 1.

Exercice 2 – Construction d'un arbre en largeur

On veut un algorithme distribué pour construire un arbre en largeur enraciné en P_i . On suppose le réseau asynchrone.

On utilise le principe de l'inondation :

L'algorithme en P_i est le suivant :

Procédure construction

envoyer $\langle\langle i \rangle\rangle$ à tous les voisins
 $nbrep := \text{nombre de voisins}$

A la réception de $\langle acq \rangle$

$nbrep := nbrep - 1$
 si $nbrep = 0$ alors fin

L'algorithme en P_j , $i \neq j$ est le suivant :

A la réception de $\langle i \rangle$ de d

si participant alors envoyer $\langle acq \rangle$ sur d
 sinon participant := vrai
 $nbrep := \text{nombre de voisins} - 1$

```

pere := d
si nbrep ≠ 0 alors envoyer <i> à tous les voisins sauf d
  sinon envoyer <acq> sur d

```

A la réception de <acq> *nbrep* := *nbrep* - 1
si *nbrep* = 0 alors envoyer <acq> sur *pere*

1. Pourquoi l'arbre ainsi construit n'est pas forcément un arbre en largeur ?
2. Modifier l'algorithme pour construire un arbre réellement en largeur enraciné en P_i (conseil : construire l'arbre niveau par niveau).
3. Comment baisser le nombre de messages ?

Correction exercice 2

1. Il suffit de prendre un triangle a, b et c . Le site a envoie la procédure de construction. b dès qu'il a reçu le message de a renvoie un message à c . Donc c va traiter le message de b avant le message de a .

2. Voici le code :

Le site P_i :

```

Niveaui := 0 ;
Envoyer(i, Niveaui+1) aux voisins ;

```

Le site P_j avec $i \neq j$:

Reception(*i*, Niveau) venant de k :

Si (*participant*) et Niveau ≤ Niveau _{j} alors

Envoyer(*ack*) à k ;

Sinon

Si (*participant*) et Niveau < Niveau _{j} alors

pere := k ;

Niveau _{j} := Niveau ;

Nbrep := Nvoisins - 1 ;

Si Nbrep ≠ 0 alors Envoyer(*i*, Niveau _{$j+1$}) aux voisins - { k } ;

Sinon Envoyer(*ack*) à k ;

Si (!*participant*) alors

```

participant := vrai ;
idem

```

3. Il faut qu'il y ait un temps borné de transmission on met donc une horloge.

Fin correction exercice 2

Exercice 3 – Construction d'un anneau virtuel

Un anneau virtuel peut être défini comme une relation d'ordre total sur l'ensemble X des sites, il peut donc être vu globalement comme une site circulaire dans laquelle chaque élément de X apparaît une fois et une seule, cette structure peut être distribuée entre les sites du réseau : il suffit que chacun d'eux soit doté d'une variable *succ_i* est son suivant dans la liste (ou dans la relation d'ordre). L'exploitation de cet anneau virtuel se fait à l'aide d'un jeton qui parcourt cette liste.

Rôle de l'initiateur *pere_α* := $α$;

Si *voisins_α* = ∅ alors

terminé, l'anneau est réduit à un seul site

Sinon soit $k \in \text{voisins}_α$

pv := k ;

envoyer explorer({*i*}, *i*) à P_k ;

A la réception de explorer(z, d) depuis P_j

succ_i := d ; *pere_i* := j

Si *voisins_i* ⊆ z alors

$R_i(j)$:= j

envoyer retour($z \cup \{i\}, i$) à P_j

Sinon

soit $k \in \text{voisins}_i - z$

$R_i(k)$:= j

envoyer explorer($z \cup \{i\}, i$) à P_k

A la réception de retour(z, d) depuis P_j

Si *voisins_i* ⊆ z alors

Si *pere_i* = i alors

P_i est l'initiateur $P_α$, l'anneau est bouclé

succ_i := d ; $R_i(pv)$:= j

Sinon $R_i(pere_i)$:= j

envoyer retour(z, d) à P_{pere}

Sinon

soit $k \in voisins_i - z$

$R_i(k) := j$

envoyer *explorer*(z, d) à P_k

1. La création de la variable $succ_i$ nécessite une information sur le suivant dans l'anneau. Que devons-nous savoir?
2. Nous supposons par la suite que chaque site P_i possède une table de routage R_i telle que :
 - $R_i(j)$ contient l'identité de son voisin auquel P_i doit transmettre le jeton lorsqu'il le reçoit de P_j .Nous devons donc définir les variables $succ_i$ et $R_i \forall i$.
 - (a) Comment affecter les variables $succ_i$?
 - (b) Comment définir les tables de routage?
 - (c) Comment l'initiateur crée sa table de routage?
3. Quel est la complexité?
4. Donner l'algorithme qui permet l'exploitation du jeton sur l'anneau virtuel

Correction exercice 3

1. Le lien entre un site de l'anneau et son successeur n'est pas forcément physique. Il en résulte que l'acheminement du jeton depuis un site P_i vers son successeur logique P_{succ} peut se faire via d'autres sites. Le passage du jeton sur un site peut avoir deux significations distinctes :
 - soit le site est le destinataire du jeton,
 - soit le site ne fait que relayer le jeton sur le chemin le conduisant à sa prochaine destination
2. Détail des variables
 - (a) Lorsqu'un processus P_i reçoit pour la première fois le jeton de parcours, il note l'identité transportée par ce dernier comme étant son successeur logique $succ_i$ sur l'anneau. En conséquence, le déplacement sur l'anneau sera exactement l'inverse du déplacement physique du jeton de parcours.
 - (b) Les tables de routage en découlent immédiatement : lorsque P_i reçoit le jeton de parcours depuis un voisin P_j et le retransmet à un voisin P_k (éventuellement le même), il établit $R_i(k) := j$.

- (c) Pour l'initiateur, la situation est un peu différente : il crée le jeton de parcours au début de l'algorithme, et le détruit à la fin. Initialement, P_α envoie le jeton à un voisin, pv , et lorsqu'il détecte la terminaison (après avoir reçu le jeton de parcours d'un de ses voisins, dv), la liaison $R_i(pv) := dv$ doit être établie ; pour cela, P_α devra mémoriser, dans une variable locale, l'identité du premier voisin auquel il envoie le jeton de parcours.

3. Voici la solution

Rôle de l'initiateur $pere_\alpha := \alpha$;

Si $voisins_\alpha = \emptyset$ alors

terminé, l'anneau est réduit à un seul site

Sinon soit $k \in voisins_\alpha$

$pv := k$;

envoyer *explorer*($\{i\}, i$) à P_k ;

A la réception de *explorer*(z, d) depuis P_j

$succ_i := d$; $pere_i := j$

Si $voisins_i \subseteq z$ alors

$R_i(j) := j$

envoyer *retour*($z \cup \{i\}, i$) à P_j

Sinon

soit $k \in voisins_i - z$

$R_i(k) := j$

envoyer *explorer*($z \cup \{i\}, i$) à P_k

A la réception de *retour*(z, d) depuis P_j

Si $voisins_i \subseteq z$ alors

Si $pere_i = i$ alors

P_i est l'initiateur P_α , l'anneau est bouclé

$succ_i := d$; $R_i(pv) := j$

Sinon $R_i(pere_i) := j$

envoyer *retour*(z, d) à P_{pere}

Sinon

soit $k \in voisins_i - z$

$R_i(k) := j$

envoyer *explorer*(z, d) à P_k

4. Voici la solution pour l'exploitation du jeton sur l'anneau virtuel

A la réception de *jeton*(adr) depuis P_j Si $adr = i$ alors

utiliser le jeton $adr := succ_i$ $k := R_i(j)$

envoyer *jeton(adr)* à P_k

Fin correction exercice 3

Exercice 4 – Construction d’un arbre « en profondeur d’abord »

Soit $G = (V, E)$ un graphe connexe quelconque représentant le réseau physique. On veut créer dans G un arbre de type *en profondeur d’abord* à partir d’un sommet *racine*. Chaque sommet maintiendra la connaissance de son *père* et de ses *fil*s grâce à un marquage des liens correspondants (la procédure *mark* permet de changer ce marquage). Au début, tous les sommets sont dans l’état *idle* et tous les liens sont marqués *nonvisite*. Voici une partie de l’algorithme pour le sommet i :

Procédure *chercher* ;

```
Si ( $\exists k : mark_i(k) = nonvisite$ ) alors
  Envoyer(<TOKEN>) sur le lien  $k$  ;
   $mark_i(k) := fils$  ;
Sinon Si ( $i = initiateur$ ) alors stop ;
  Sinon Envoyer(<TOKEN>) sur le lien  $k$  tel que  $mark_i(k) = pere$  ;
```

Initialisation (* à exécuter seulement par l’initiateur *) :

```
Si ( $etat_i = idle$ ) alors
   $etat_i := decouvert$  ;
  chercher ;
Pour tout ( $k : mark_i(k) = visite$  ou  $mark_i(k) = nonvisite$ )
  Envoyer (<VISITED>) sur le lien  $k$  ;
```

Lors de la réception de <VISITED> sur le lien j

```
Si ( $mark_i(j) = nonvisite$ ) alors  $mark_i(j) := visite$  ;
```

Lors de la réception de <TOKEN> sur le lien j

...

1. Ecrire la procédure de réception de <TOKEN>.
2. Donner un exemple d’exécution.

3. On donne maintenant une autre procédure de réception de <VISITED> :

Lors de la réception de <VISITED> sur le lien j

```
Si ( $mark_i(j) = nonvisite$ ) alors  $mark_i(j) := visite$  ;
Si ( $mark_i(j) = fils$ ) alors
   $mark_i(j) := visite$  ;
  chercher ;
```

A quoi sert le rajout par rapport à la précédente version?

4. Evaluer la complexité en nombre de messages.

Correction exercice 4

1. Au début de l’algorithme tous les sommets sont *idle*. On suppose que seul l’initiateur se réveille et exécute *Initialisation*. Tous les sommets ont leurs liens à *nonvisite*. Lors de la réception du message <TOKEN>, le sommet récepteur n’ayant jamais reçu le message doit marquer son père, dire qu’il est découvert, poursuivre l’exploration (exécuter la procédure *chercher*) et dire qu’il est visité à tous ses voisins. Si le récepteur a déjà été visité, lorsqu’il reçoit un <TOKEN> de son *fil*s cela indique que son fil

Lors de la réception du message <TOKEN> sur le lien j

```
Si ( $etat_i = idle$ ) alors
   $mark_i(j) := pere$  ;
   $etat_i := decouvert$  ;
  chercher ;
Pour tout ( $k : mark_i(k) = visite$  ou  $mark_i(k) = nonvisite$ )
  Envoyer (<VISITED>) sur le lien  $k$  ;
```

Sinon

```
Si ( $mark_i(j) = nonvisite$ ) alors  $mark_i(j) := visite$  ;
Si ( $mark_i(j) = fils$ ) alors chercher ;
```

2. La liste des token est : $a, b, c, b, d, e, d, b, a, e, a$ pour un graphe composé d’un cycle a, b, d, e et b admet un successeur c .

Question 3.

Le rajout permet de prendre en compte le cas des messages $\langle VISITED \rangle$ très lents. Supposons que x reçoive le $\langle TOKEN \rangle$ pour la première fois. D'après la procédure, il cherche et envoie un message $\langle VISITED \rangle$ à tous ses voisins, dont y . Ensuite il poursuit sa recherche en envoyant le $\langle TOKEN \rangle$ à a qui le renvoie à b qui le renvoie à y , avant que y n'ait reçu le $\langle VISITED \rangle$ de x . Dans ce cas, y peut poursuivre sa recherche vers x en envoyant un $\langle TOKEN \rangle$ à x . Ensuite y reçoit $\langle VISITED \rangle$. Sans la modification, y ne poursuivrait pas l'exploration (voir illustration figure 2) et le processus serait bloqué. Cependant, cette nouvelle version ne marche encore pas.

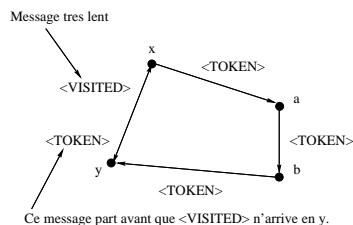


FIG. 2 – Illustration de l'effet d'un message $\langle VISITED \rangle$ lent.

Le deuxième cas pathologique est le suivant et débute exactement comme le premier : x reçoit le $\langle TOKEN \rangle$, le fait passer à a , puis a à b puis b à y . Dans notre situation, le sommet y reçoit $\langle TOKEN \rangle$ sans encore avoir eu $\langle VISITED \rangle$. Il va alors envoyer $\langle TOKEN \rangle$ à x puis recevoir $\langle VISITED \rangle$ de x . Le sommet y marque son lien yx comme *visite* et peut continuer l'exploration. A partir de là les deux versions sont différentes. Maintenant, le message $\langle TOKEN \rangle$ envoyé (à tort) par y à x est lui aussi très lent. C'est-à-dire qu'il n'arrive en x qu'après le message $\langle TOKEN \rangle$ produit par la phase de backtrack qui suit le chemin y, b, a, x . Le premier $\langle TOKEN \rangle$ est reçu par x qui doit donc poursuivre l'exploration. Pour cela, il peut décider de la poursuivre par le lien xy en envoi donc un message $\langle TOKEN \rangle$ sur le lien xy qu'il marque comme fils. Le lien xy voit donc se croiser deux messages $\langle TOKEN \rangle$.

- y reçoit le $\langle TOKEN \rangle$ de x sur un lien marqué *visite*. D'après l'algorithme il ne fait rien.
- x reçoit le $\langle TOKEN \rangle$ de y sur un lien marqué *fils*. D'après l'algorithme il poursuit la recherche.

Du coup, x a deux fils, a et y et on a un *circuit* dans le graphe en construction.

Ce problème vient du fait que l'algorithme, tel qu'il est décrit, ne prend pas en compte le lien sur lequel le sommet (ici x) reçoit le message $\langle TOKEN \rangle$. Il faut rajouter des

indications pour être sûr que l'on reçoit bien un message $\langle TOKEN \rangle$ de backtrack de son propre fils et pas un $\langle TOKEN \rangle$ de quelqu'un d'autre. On va donc distinguer les messages $\langle TOKEN \rangle$ (notés $\langle TOKEN(DESC) \rangle$) de descente et les messages $\langle TOKEN(BACK) \rangle$ de backtrack (notons que cette information ne nécessite qu'un bit pour être codée dans le message). Nous pouvons alors proposer la nouvelle version prenant en compte les changements.

Lors de la réception du message $\langle TOKEN(DESC) \rangle$ sur le lien j

Si ($etat_i = idle$) alors

$mark_i(j) := pere;$

$etat_i := decouvert;$

chercher ;

Pour tout ($k : mark_i(k) = visite$ ou $mark_i(k) = nonvisite$)

Envoyer ($\langle VISITED \rangle$) sur le lien k ;

Sinon

Si ($mark_i(j) = nonvisite$) alors $mark_i(j) := visite$;

Si ($mark_i(j) = fils$) alors /* C'est un cas anormal pour $\langle TOKEN(DESC) \rangle$ */

$mark_i(j) := visite$;

chercher ;

Lors de la réception du message $\langle TOKEN(BACK) \rangle$ sur le lien j

Si ($mark_i(j) = fils$) alors chercher ;

Cette modification résout le problème mentionné plus haut concernant l'obtention d'un circuit. En effet, avec la nouvelle version, y a envoyé un $\langle TOKEN(DESC) \rangle$ à y et idem de y à x . Regardons comment réagissent les deux sommets :

- y reçoit un $\langle TOKEN(DESC) \rangle$ sur un lien qui est marqué *visite* (à cause du message $\langle VISITED \rangle$ qu'il a reçu plus tôt de x). En appliquant l'algorithme il ne fait donc rien.
- x reçoit un $\langle TOKEN(DESC) \rangle$ sur un lien qui est marqué *fils* (il l'a marqué après avoir envoyé $\langle TOKEN(DESC) \rangle$ à y). D'après l'algorithme, il va marquer *visite* le lien et va exécuter la fonction chercher, ce qui est conforme à ce que l'on attend.

Question 4.

Complexité en nombre de messages. On peut montrer qu'au plus deux $\langle TOKEN() \rangle$ et qu'un plus deux $\langle VISITED \rangle$ peuvent traverser chaque lien. On échange donc au plus $4|E|$ messages. Une analyse plus raffinée montre que l'on en échange un peu moins en fait.

Fin correction exercice 4

Exercice 5 – L’algorithme de Dijkstra en séquentiel

Cet exercice a pour but de vous remettre en mémoire l’algorithme de Dijkstra. Voici l’algorithme de Dijkstra :

Algorithm 1 L’algorithme de Dijkstra

```

 $d(s) := 0; \text{pred}(s) := 0$ 
 $d(i) := +\infty; \text{pour tout } i \neq s$ 
 $S = \emptyset$ 
while  $S \neq X$  do
  choisir  $i \in \bar{S}$  avec  $d(i)$  minimum
   $S = S \cup \{i\}$ 
  for chaque arc  $(i, j)$  do
    if  $(i, j)$  est un raccourci then
      l'emprunter
    end if
  end for
end while

```

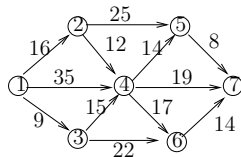
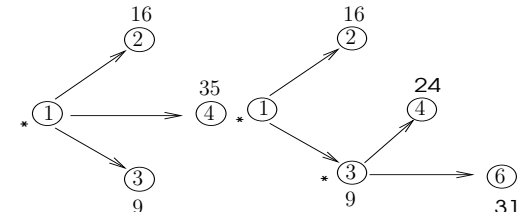


FIG. 3 – Graphe

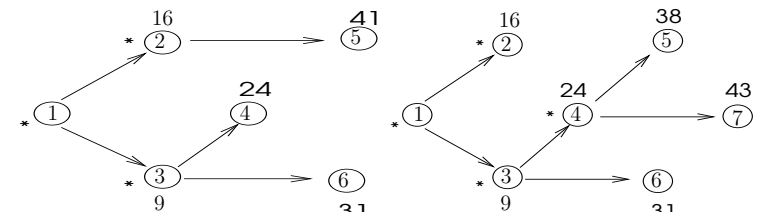
1. Appliquer cet algorithme sur le graphe donné par la figure 3 à partir du sommet 1.
2. Démontrer que l’algorithme de Dijkstra résout le problème des plus courts chemins en temps $\mathcal{O}(n^2)$.

Correction exercice 5

1. Solution de l’application.
2. À la i -ième étape on sélectionne le sommet de \bar{S} ayant la plus petite distance. La taille de \bar{S} est alors égale à $n - i$, et cela nécessite donc $n - i$ opérations. À la i -ième étape on corrige également (éventuellement) tous les successeurs du sommet x_i



(a) étape 1. Nous avons $S = \{1\}$. Le sommet 1 a été choisi car c’est lui qui avait la plus petite étiquette, égale à 0. Les 4 ayant respectivement les étiquettes raccourcis (1,2), (1,3) et 16,9 et 35. Les raccourcis (3,4) et (1,4) ont été empruntés. (3,6) ont été empruntés.



(c) étape 3. Nous avons $S = \{1, 3, 2\}$. Le sommet 2 a été choisi car c’est lui qui avait la plus petite étiquette, égale à 16, parmi les sommets 2,4 et 6 ayant respectivement les étiquettes 16,24 et 31. (d) étape 4. Nous avons $S = \{1, 3, 2, 4\}$. Le sommet 4 a été choisi car c’est lui qui avait la plus petite étiquette, égale à 24, parmi les sommets 4,5 et 6 ayant respectivement les étiquettes 24,31 et 41.

FIG. 4 – Illustration de l’algorithme de Dijkstra

choisi. Cela nécessite $|\Gamma^+(x_i)|$ opérations. On a donc au total $\sum_{i=0}^{n-1} (n - i + |\Gamma^+(x_i)|) = \sum_{i=0}^{n-1} (n - i) + \sum_{i=0}^{n-1} |\Gamma^+(x_i)| = \frac{n(n-1)}{2} + m = \mathcal{O}(n^2)$ opérations.

Fin correction exercice 5

Exercice 6 – Maintien des plus courts chemins dans un graphe connexe quelconque.

Dans cet exercice nous proposons d’écrire un algorithme réparti qui va construire entre chaque paire de sommets de $G = (V, E)$ un chemin de plus courte distance. Les communications se font entre voisins et sont asynchrones. Chaque sommet connaît la taille du graphe (n

sommets) et les liens de communication sont *FIFO*. La construction des plus courts chemins est faite par "apprentissage" du réseau. Chaque sommet u maintient :

- Un tableau D_u dont la case $D_u[v]$ est la distance estimée par u de $d_G(u, v)$.
- Un tableau $Sortie_u$ dont l'entrée $Sortie_u[v]$ indique le voisin w de u qui est sur un plus court chemin de u à v .
- Un tableau $estimat_u$ dont l'entrée $estimat_u[w, v]$, avec $w \in voisins_u$, est l'estimation par u de la distance $d_G(w, v)$.

Voici une partie de l'algorithme pour un sommet u quelconque :

Procédure recalcul_Dist(v)

...

Procédure Initialisation

```
Pour tout ( $w \in voisins_u, v \in V$ ) faire  $estimat_u[w, v] := n$  ;
Pour tout ( $v \in V$ ) faire
     $D_u[v] := n$  ;
     $Sortie_u[v] := NIL$  ;
 $D_u[u] := 0$  ;
 $Sortie_u[u] := local$  ;
Pour tout ( $w \in voisins_u$ ) faire Envoyer( $\langle MADIST, u, 0 \rangle$ ) à  $w$  ;
```

Lors de la réception de $\langle MADIST, v, d \rangle$ de w ($w \in voisins_u$)

```
 $estimat_u[w, v] := d$  ;
recalcul_Dist( $v$ ) ;
```

1. Ecrire la procédure *recalcul_Dist*.
2. Pourquoi, dans un graphe dont la topologie ne change pas, l'algorithme s'arrête au bout d'un temps fini? Quelles sont alors les valeurs des divers paramètres? Proposez une fonction de routage basée sur les résultats de l'algorithme précédent.
3. Que peut on faire en cas de pannes de liens sachant qu'un sommet peut être prévenu de la panne d'un de ses liens adjacent par une interruption?

Correction exercice 6

1. Cette méthode est basée sur l'algorithme de Dijkstra pour le calcul des plus courts

chemins. On affine peu à peu la connaissance des distances que l'on a du réseau.

Procédure recalcul_Dist(v)

```
Si ( $v = u$ ) alors
     $D_u[v] := 0$  ;
     $Sortie_u[v] := local$  ;
Sinon
     $d := 1 + \min\{estimat_u[w, v] : w \in voisins_u\}$  ;
    Si ( $d < n$ ) alors
         $D_u[v] := d$  ;
         $Sortie_u[v] := w$  ; avec  $w$  tel que  $d = 1 + estimat_u[w, v]$ .
    Sinon
         $D_u[v] := n$  ;
         $Sortie_u[v] := NIL$  ;
Si ( $D_u[v]$  a changé) alors
    Pour tout ( $w \in voisins_u$ ) faire Envoyer( $\langle MADIST, v, D_u[v] \rangle$ ) ;
```

2. L'algorithme s'arrête car au bout d'un certain temps $D_u[v]$ ne va plus changer et les messages $\langle MADIST, .. \rangle$ ne seront plus envoyés (grâce au dernier test Si ($D_u[v]$ a changé) ...).

A la fin, on aura, $D_u[v] = d_G(u, v)$ et $Sortie_u[v]$ est le voisin w de v qui est sur le plus court chemin entre u et v . En fait à chaque "étape", $D_u[v] \leq d_G(u, v)$ et $estimat_u[w, v] \leq d_G(w, v)$. A la fin ces quantités sont égales.

La fonction de routage associée : un message arrivant au sommet u , à destination du sommet v sera envoyé vers le sommet $Sortie_u[v]$ pour poursuivre sa route (ou sur la sortie locale si $u = v$).

3. En cas de panne de lien(s), il faut recalculer les distances avec la procédure *recalcul_Dist*.

Lors de la détection d'une panne sur le lien u, w

```
 $voisins_u := voisins_u - \{w\}$  ;
Pour tout ( $v \in V$ ) faire recalcul_Dist( $v$ ) ;
```

En cas d'ajout d'un lien.

Lors de la détection d'un ajout d'un lien u, w

```
 $voisins_u := voisins_u \cup \{w\}$  ;
Pour tout ( $v \in V$ ) faire
     $estimat_u[w, v] := n$  ;
    Envoyer( $\langle MADIST, v, D_u[v] \rangle$ ) à  $w$  ;
```

contenant aucun sommet de M .
retourner (T_2) ;

Fin correction exercice 6

Dans les exercices suivants nous allons représenter un système par un graphe $G = (V, E)$. L'ensemble V des *sommets* représente les sites. On a une arête $[u, v] \in E$ entre le sommet u et le sommet v s'il existe un lien de communication entre u et v dans le réseau sous-jacent entre les sites représentés par u et v . Nous ne considérons ici que des *graphes connexes*.

Exercice 7 – Structures de réseaux bis Dans un système donné représenté par un graphe $G = (V, E)$ on veut interconnecter les sommets d'un ensemble $M \subset V$ par un arbre. Pour des raisons d'économie et d'encombrement on veut utiliser le moins d'arêtes possibles. On cherche donc à construire un arbre $T_0 = (V_{T_0}, E_{T_0})$ tel qu'il vérifie les deux conditions suivantes.

- $M \subseteq V_{T_0} \subseteq V$ et $E_{T_0} \subseteq E$. Un tel arbre T_0 est dit *arbre de Steiner de M dans G* .
- T_0 a le plus petit nombre possible d'arêtes parmi tous les arbres de Steiner de M possibles.

Si T_0 vérifie les deux conditions il est dit *arbre de Steiner optimal de M dans G* . Le nombre d'arêtes d'un arbre de Steiner de M sera appelé son *poide*. Dans les questions suivantes, on supposera que G et M sont donnés et que w_0 est le poide d'un arbre de Steiner optimal de M dans G .

1. Donnez une borne inférieure de w_0 . Dans quels cas cette borne est-elle atteinte?
2. Donnez une borne supérieure de w_0 . Dans quels cas cette borne est-elle atteinte?

Considérons l'algorithme (non réparti) suivant pour construire un arbre de Steiner de M dans G .

Procédure Steiner(G, M)

Choisir un sommet quelconque $r \in M$;

$S := \{r\}$; $G_1 = (\{r\}, \emptyset)$

tant que ($S \neq M$) faire

 Choisir un sommet $u \in M$ le plus près possible d'un sommet v de

S ;

 Ajouter dans G_1 un plus court chemin de G entre u et v ;

$S := S \cup \{u\}$;

Construire T_1 l'arbre obtenu en supprimant tous les cycles de G_1 ;

Construire T_2 l'arbre obtenu en supprimant tous les sous-arbres de T_1

ne

3. Faire une exécution de cet algorithme dans un graphe G quelconque et un ensemble M quelconque. Montrez que le résultat de cet algorithme est un arbre de Steiner de M dans G . Montrez qu'il ne donne pas toujours un arbre de Steiner optimal.
4. Dans quels cas cet algorithme peut-il donner à coup sûr un arbre de Steiner optimal?
5. Soit $W(T)$ le poide de l'arbre obtenu en appliquant l'algorithme et w_0 le poide de l'arbre de Steiner optimal. Montrez que $\frac{W(T)}{w_0} \leq 2$.

Correction exercice 7

1. Si $m = |M|$ alors $m - 1 \leq w_0$ car il faut que T_0 soit un arbre contenant au moins les m sommets. On a $w_0 = m - 1$ lorsque le graphe G' suivant est connexe : $V' = M$ et $E' = \{[u, v] : u, v \in M\}$.
2. Si $n = |V|$ alors $w_0 \leq n - 1$ car tout arbre couvrant tous les sommets de G (ayant donc $n - 1$ arêtes) est un arbre de Steiner de M dans G . On a $w_0 = n - 1$ lorsque $M = V$.
3. On peut faire une exécution dans la roue pour montrer l'algorithme et le fait qu'il n'est pas optimal. L'algorithme ressort bien un arbre de Steiner de M car on obtient bien un arbre couvrant tous les sommets de M dans G .
4. On obtient un arbre de Steiner optimal lorsque l'on est dans les conditions de la question 1. Dans ce cas, les distances choisies seront 1 à chaque étape et de manière gloutonne on va construire à coup sûr un arbre ayant comme uniques sommets mes sommets de M .
5. Voir la correction séparée.

Fin correction exercice 7

Exercice 8 – Le problème d'apprentissage des identités

Partie I

Chaque site P_i est doté d'une identité i unique dans tout le système. Chaque site a donc une connaissance globale a priori : son identité est différente de celle de ses voisins. Nous nous intéressons à la détermination des identités pour tout i des voisins du site P_i . Ceci est facilement réalisé par un protocole d'échange d'identités entre tout couple de sites connectés

par une ligne bi-directionnelle. Un ou plusieurs peuvent décider de un tel échange; celui-ci se généralise alors à tout le réseau. Chaque site P_i est doté du contexte suivant :

```
var : participant(i) : booléen initialisé à faux ;
      canaux(i) : ensemble de lignes initialisé à
                {lignes qui relie P_i à d'autres sites } ;
      voisins(i) : ensemble de noms initialisé à vide ;
```

1. Donner l'algorithme associé au problème décrit ci-dessus et justifier-le.
2. Donner la complexité en nombre de messages.

Partie II

Nous supposons que chaque site P_i connaît ses voisins. Apprendre le réseau consiste pour un site donné à apprendre le graphe G associé c'est-à-dire l'ensemble des sites (sommets de G) et l'ensemble des lignes (arêtes de G). Nous supposons qu'un ou plusieurs sites décident de lancer l'algorithme d'apprentissage simultanément. Dans la suite nous allons considérer les éléments suivant :

```
var : participant(i) : booléen initialisé à faux ;
      voisins(i) : ensemble de noms initialisé à
                {identité des voisins de P_i};
      sitesconnues(i) : ensemble de nom initialisé à {i} ;
      lignesconnues(i) : ensemble de couples de noms initialisé à
                {(i,x) : x élément de l'ensemble voisins(i) ;
```

et le message *identité(j,v)* qui véhicule l'identité du site j , et l'ensemble v de celles de ses voisins.

1. Comment un site P_i sait-il qu'il connaît tout le réseau? On introduit pour cela le prédicat *terminé_i*. Compléter la propriété sur le prédicat *terminé_i* et justifier votre réponse :

$$\text{terminé}_i \equiv ((x, y) \in \text{lignesconnues}_i \rightarrow).$$
2. Donner l'algorithme associé et justifier-le.
3. Donner la complexité au pire sur le nombre de messages.

Partie III

Nous nous intéressons maintenant à la recherche de chemins dans le réseau. Le problème est le suivant : il consiste, pour chaque site P_i , à apprendre quelle est sa distance à chacun des autres sites, c'est-à-dire la plus petite distance en nombre d'arêtes. Il s'agit du problème

classique des plus court chemins dans un graphe, chaque arête étant valué par 1. Une solution consiste à utiliser l'algorithme de la **Partie II**. Chaque site p_i une fois le réseau connu, peut calculer pour chaque site P_j la plus courte distance (en nombre de lignes à franchir) de P_i à P_j . Il peut utiliser les informations obtenues pour effectuer un routage optimal des messages qui seront ensuite émis vers les autres sites.

Nous allons en proposer un autre algorithme qui ne nécessite pas la connaissance du réseau. Nous supposons que chaque site connaît une majorant *maxn* sur le nombre de sites présents dans le système, du fait qu'il s'agit d'un problème de nature globale.

1. Donner les variables nécessaires à la résolution du problème et leur signification. Vous pourrez utiliser la variable *distance_i* (tableau de taille *maxn* telle que *distance_i[j]* indiquera à P_i sa plus courte distance à P_j . Si $j \in \text{voisin}_i$ alors *distance_i[j]* = 1.
2. Donner l'algorithme.
3. Quels sont les solutions possibles pour résoudre le problème de la terminaison de l'algorithme.

Correction exercice 8

Partie I

1. Voici l'algorithme :

Lors de la décision de provoquer un échange
participant_i := vrai;
 $\forall c \in \text{canaux}_i$: envoyer *identité(i)* sur c

Lors de la réception de *identité(j)* sur cl
associer(cl, j)
 si non *participant_i* alors
 participant_i := vrai;
 $\forall c \in \text{canaux}_i$: envoyer *identité(i)* sur c

2. Sous les hypothèses générales, deux messages *identité* transitent en sens inverse sur chaque ligne : la complexité en nombre de messages est donc $2e$ où e est le nombre de lignes.

Partie II

1. $\text{terminé}_i \equiv ((x, y) \in \text{lignesconnues}_i \rightarrow (x \in \text{sitesconnues}_i \text{ et } y \in \text{sitesconnues}_i)).$

2. Voici l'algorithme : Lors de la décision d'apprendre le réseau

$participant_i := vrai;$
 $\forall x \in voisins_i : \text{envoyer } identité(i, voisins_i) \text{ à } P_x$

Lors de la réception de $identité(j, v)$ de P_k

si *non participant*_{*i*} alors
 $participant_i := vrai;$
 $\forall x \in voisins_i : \text{envoyer } identité(i, voisins_i) \text{ à } P_x$
 si $j \notin sitesconnues_i$ alors
 $sitesconnues_i := sitesconnues_i \cup \{j\}$
 $lignesconnues_i := lignesconnues_i \cup \{(j, x) : \forall x \in v\}$
 $\forall x \in voisins_i - \{k\} : \text{envoyer } identité(j, v) \text{ à } P_x;$
 si *termin*_{*i*} alors P_i connaît tout le réseau

3. Un message $identités(i, voisins_i)$ au moins et 2 au plus sont transmis sur une ligne donnée (un dans chaque sens s'il s'agit de 2); s'il y a n sites et e lignes la complexité en nombre de messages de l'algorithme est au pire de $2ne$ messages.

Partie III

1. les variables sont :

- var $distance(i)$: tableau $[1..maxn]$ de entier positif ou nul initialisé à $maxn$ pour tout les indices j différent de i et pour i c'est initialisé à 0;
- $routage(i)$: tableau $[1..maxn]$ de 0 à $maxn$ initialisé à 0

Connaitre les valeurs de splus courts chemins est une chose, pourvoir les exploiter en est une autre. Pour cela tout site P_i doit connaitre lequel de ses voisins est son meilleur voisins pour atteindre un site donné P_j ; cette information est nécessaire pour effectuer un routage optimal des messages. une fois les valeurs des pus courts chemins calculées, $routage_i[j]$ donne le meilleur voisin de P_i pour atteindre P_j , si ce site existe dans le réseau. Ce tableau ne sert que de manière interne à P_i .

Le principe de l'algorithme est le suivant :

Le principe sur lequel repose l'algorithme est simple et s'apparente à un calcul itératif. Initialement un ou plusieurs site diffuse son tableau des distances à ses voisins (c'est à dire sa connaissance initiale). Lorsqu'un site reçoit une telle information il met à jour son contexte et si celui-ci a été modifié (il a appris des distances plus courtes que celles qu'il connaissait préalablement) il diffuse à ses voisins son nouveau tableau des distances (c'est à dire sa nouvelle connaissance). Tout site recevant un message obéit à ce comportement. On peut voir conceptuellement des étapes de calcul : un site

apprend d'abord quels sites sont à une distance de 1, puis ceux qui sont à une distance de 2, et ainsi de suite. Tous les sites étant à une distance finie les uns des autres, $maxn$ constitue un majorant du nombre de telles étapes.

Le principe selon lequel les sites coopèrent étant établi, il reste à énoncer le principe selon lequel les sites mettent à jours leur contexte lors de la reception d'un message. ces messages sont d'un unique type *majdist* et véhiculent un tableau d . Considérons le site P_i recevant un tel message en provenance de son voisin P_j . d d'une part représente la valeur de distance_{*j*} lors de l'envoi du message, et d'autre part informe P_i que cette valeur est une nouvelle information que P_j ne lui a pas encore transmis. Pour tout site potentiel P_k ($1 \leq k \leq maxn$), le site P_i va tester si le message reçu lui permet de déterminer une distance plus courte vers P_k , auquel cas il prend en compte cette nouvelle distance et positionne un indicateur qui, une fois tous les sites examinés, lui signalera qu'il a acquis des nouvelles informations et en conséquence il doit les transmettre à ses voisins.

2. Voici l'algorithme :

Lors de la décision de calculer les valeurs de splus court chemins

$\forall x \in voisins_i ; \text{envoyer } majdist(distance_i) \text{ à } P_x$

Lors de la réception de $majdist(d)$ de P_j

$modif := faux$
 pour k depuis 1 jusqu'à $maxn$
 si $distance_i[k] > 1 + d[k]$
 alors $routage_i := [k] := j ; distance_i[k] := 1 + d[k] ; modif := vrai$
 si *modif* alors
 $\forall x \in voisins_i ; \text{envoyer } majdist(distance_i) \text{ à } P_x$

3. Cet algorithme ne permet pas à un site de savoir si l'algorithme est non terminé. Plusieurs solutions sont envisageables si cette connaissance est nécessaire. Une solution consisterait par exemple à utiliser l'algorithme d'apprentissage du réseau vu précédemment et à doter les sites d'un algorithme séquentiel ad'hoc. Une solution plus orthodoxe consiste à élaborer un algorithme semblable au précédent mais dans lequel des étapes de calcul sont fortement synchronisées. Tous les sites démarrent l'algorithme (c'est-à-dire l'étape 1) simultanément ; ceci est facilement réalisé : le ou les sites qui ont démarré le calcul provoquent par la diffusion aux voisins de leur premier message une *réaction en chaîne* qui va activer tous les sites et dès qu'un site est activé, il devient participant et commence par diffuser son premier message à ses voisins. C'est durant l'étape p que chaque site P_i apprend quels sites sont à distance minimale de p ; une

telle étape n'est démarrée que lorsque l'étape $p - 1$ est terminée. Un site sait que l'étape $p - 1$ est terminée lorsque durant cette étape il a reçu un message de chacun de ses voisins. La prise en compte de ces messages lui permet de passer à l'étape suivante s'il a appris du nouveau ou dans le cas contraire de conclure qu'il n'apprendra plus rien (rappelons que le réseau est connexe) ; l'algorithme est alors terminé en ce qui le concerne, il peut en informer ses voisins et écartera ensuite les messages reçus.

Fin correction exercice 8

Exercice 9 – Election dans un graphe quelconque

On considère un réseau quelconque (connexe) dans lequel les lignes sont bidirectionnelles et fiables ; chaque processus ne connaît pas ses voisins et il connaît par leurs identités. Au cours du calcul un processus ne doit pas apprendre d'informations globales (telle que la structure du réseau par exemple) qu'il pourrait ensuite utiliser ; (une telle contrainte favorise le rétablissement après panne). Il s'agit de concevoir un algorithme distribué, qui peut être démarré par un ou plusieurs processus, tel qu'à la fin de l'algorithme un processus unique ait été élu par les autres et que chacun des autres en connaisse l'identité et connaisse également lequel de ses voisins permet d'atteindre le processus élu.

1. Quel maximum sous-graphe (au sens de l'inclusion) pouvons nous utiliser pour reformuler le problème ?
2. Un des problèmes délicats à résoudre est le fait que l'algorithme peut être démarré par un nombre quelconque de processus. Proposer une idée pour bloquer l'exploration venant d'un processus ayant une identité i plus faible que l'exploration d'un processus j avec $i < j$.

Chaque processus P_i est doté du contexte suivant :

- $voisins_i$, initialisée à $\{identités\ des\ voisins\ de\ P_i\}$;
- $etat_i$: (initial, candidat, battu, élu), initialisé à initial. Cette variable donne l'état courant de P_i ; à la fin de l'algorithme une seule de ces variables a la valeur *élu*, les autres ayant la valeur *battu*
- $pgvu_i$, entier initialisé à i . Il s'agit de la plus grande identité qu'ait jamais vu passer P_i ; à la fin de l'algorithme elle contiendra la plus grande des identités.
- $pere_i$, entier initialisé à *nil*
- $filis_i$, ensemble d'entiers initialisé à \emptyset . Ces deux variables servent à placer P_i

3. Donner la procédure lancer une élection. Pour un site i donné, à quel voisin j va envoyer la procédure lancer une élection.
4. Donner la procédure réception d'explorer.

5. Donner la procédure réception de rebrousser.
6. Donner la procédure réception conclure.
7. Quel est la complexité de votre algorithme en nombre de messages ?

Correction exercice 9

1. Le problème peut être reformulé de la façon suivante : il s'agit en de construire un arbre de racine le processus doté de la plus grande identité ; à la fin de l'algorithme chacun des processus doit connaître la racine, celui de ses voisins qui permet de l'atteindre (son père dans l'arbre) et ceux de ses voisins qui sont ses fils dans l'arbre.
2. Un des problèmes délicats à résoudre est le fait que l'algorithme peut être démarré par un nombre quelconque de processus. Le principe sur lequel nous allons construire l'algorithme est le suivant. Pour un processus P_i lancer une élection consiste à lancer une exploration du réseau dont le but est de visiter tous les processus en établissant des chemins de ceux-ci vers lui-même. Comme l'algorithme peut être démarré par plusieurs processus, plusieurs explorations peuvent être simultanément en cours. Au terme de l'algorithme il est nécessaire que seule l'exploration issue du processus doté de la plus grande identité d'une part se soit terminée correctement (i.e. ait visité tous les processus) et d'autre part que celle-ci soit la dernière exploration prise en compte par les autres processus pour l'établissement de l'arbre. On peut identifier une exploration de façon unique par l'identité du processus dont elle est issue. Les deux objectifs ci-dessus seront atteints dès lors que tout processus P_i visité par exploration de poids j la stoppe sans la prendre en compte s'il a été visité par une exploration d'identité k avec $k > j$; ou bien n'ayant jamais visité il trouve $i > j$, auquel cas il lance une exploration d'identité i s'il ne l'a pas déjà fait. Dans les autres cas il prend en compte l'exploration de poids j et la fait progresser.
3. Deux types d'exploration issus d'un processus P_i sont envisageables : une exploration en parallèle (la progression se fait vers tous les voisins) ou une exploration en profondeur d'abord (la progression se fait vers les voisins les uns après les autres).
4. L'exploration en profondeur va se faire avec des messages du type *explorer*. un tel message va véhiculer trois informations : k l'identité du processus qui a lancé l'exploration correspondante, z l'ensemble des identités des processus visités par ce message, et s l'ensemble des identités des processus voisins immédiats des processus de z qui n'ont pas encore été atteints par cette exploration. Au cours de sa progression le message

$explorer(k, z, s)$ voit ses champs z et s se modifier et tant que $s \neq \emptyset$ l'exploration n'est pas terminée. (en termes de graphes s représente les sommets pendants) Comme on le voit la condition d'arrêt de l'exploration d'identité maximum porte sur un champ destiné à contenir des identités (les autres explorations auront été stoppées avant).

L'exploration se faisant par un parcours en profondeur on introduit les messages $rebrousser(k, z, s)$ qui permettent à une exploration qui n'a pas atteint tous les processus (on a alors $s \neq \emptyset$) qui peut progresser (elle n'a pas encore été stoppée par un processus) mais ne peut le faire à partir du processus actuellement visité de repartir à partir d'un processus visité qui possède un voisin appartenant à s).

Les principes de l'algorithme d'élection étant établis (progrèsion et stoppage d'explorations) et un modèle de parcours en profondeur informé ayant été défini pour réaliser les explorations, l'algorithme consiste à rassembler le tout.

Si les processus doivent être informés de la terminaison de l'élection il est nécessaire d'introduire le type de message *conclure* qui lancé par le processus qui trouve $s = \emptyset$, leur indique que l'élection est terminée et qu'en conséquence les variables indiquant la plus grande identité, leur père et leur fils dans l'arbre ont leur s valeurs finales.

Chaque processus P_i est doté du contexte suivant :

- $voisins_i$, initialisé à $\{identités\ des\ voisins\ de\ P_i\}$;
- $etat_i$: (initial, candidat, battu, élu), initialisé à initial. Cette variable donne l'état courant de P_i ; à la fin de l'algorithme une seule de ces variables a la valeur *élu*, les autres ayant la valeur *battu*
- $pgvu_i$, entier initialisé à i . Il s'agit de la plus grande identité qu'ait jamais vu passer P_i ; à la fin de l'algorithme elle contiendra la plus grande des identités.
- $pere_i$, entier initialisé à *nil*
- $fils_i$, ensemble d'entiers initialisé à \emptyset . Ces deux variables servent à placer P_i

5. Lors de la décision de lancer-exploration

```

soit  $x := \max(voisins_i)$ 
 $etat_i := candidat$ 
 $pere_i := nil$ 
 $fils_i := \{x\}$ 
envoyer( $explorer(i, \{i\}, voisins_i - \{x\}$ ) à  $P_x$ 

```

Lors de la décision de lancer une élection

```

Si  $etat_i = initial$  alors lancer-exploration

```

6. Lors de la réception de $explorer(k, z, s)$ de P_j

```

Si  $pgvu_i > k$  alors si  $etat_i := initial$  alors lancer-exploration

```

```

Sinon  $pgvu_i < k$  alors

```

```

 $etat_i := battu ; pgvu_i := k ; pere_i := j ;$ 

```

```

Soit  $y := voisins_i - z$ 

```

```

Si  $y := \emptyset$  alors

```

```

 $fils_i := \emptyset$ 

```

```

Si  $s = \emptyset$  alors envoyer(conclure,  $P_j$ )

```

```

Sinon envoyer( $rebrousser(x, z \cup \{i\}, s)$ ,  $P_j$ )

```

```

Sinon soit  $x = \max(y)$ ;  $fils_i := \{x\}$ ;  $explorer(k, z \cup \{i\}, s \cup y - \{x\})$ ,  $P_x$ )

```

7. Lors de la réception de $rebrousser(k, z, s)$ de P_j

```

Si  $pgvu_i = k$  alors soit  $y := voisins \cap s$ ;

```

```

Si  $y = \emptyset$  alors envoyer( $rebrousser(x, z, s)$ ,  $P_{pere_i}$ )

```

```

Sinon soit  $x = \max(y)$ ;  $fils_i := fils_i \cup \{x\}$ , ( $explorer(k, z, s - \{x\}$ ) à  $P_x$ )

```

8. Lors de la réception de *conclure* de P_j

```

Si  $pgvu_i = i$  alors  $etat_i := élu$ 

```

```

 $\forall x \in ((fils_i \cup pere_i) - \{j\})$  envoyer(conclure,  $P_x$ )

```

9. La complexité en nombre de message est $O(n^2)$.

10. Sur le plan des algorithmes distribués, l'algorithme présenté est intéressant à deux titres. D'une part par la seule connaissance globale qu'ont les processus qui est constituée de l'identité de leurs voisins directs, et d'autre part par l'utilisation faite de cette connaissance. Les champs z et s des messages évitent d'effectuer des parcours totalement aveugles et le champ s permet de détecter la fin du parcours (relatif à la plus grande identité) « au plus tôt ». Notons de plus que s joue ici en distribué le même rôle que celui que joue la pile dans un contexte séquentiel.

Fin correction exercice 9

Exercice 10 – Rappel en théorie des graphes

Une structure de données simple pour représenter un graphe est la matrice d'adjacence M . Pour obtenir M , on numérote les sommets du graphe de façon quelconque. $X = \{x_1, \dots, x_n\}$. M est une matrice carrée $n \times n$ dont les coefficients sont 0 et 1 telle que $M_{i,j} = 1$ si $(x_i, x_j) \in E$ et $M_{i,j} = 0$ sinon. Démontrer la proposition suivante : **Proposition : Soit**

M^p la puissance p -ième de la matrice M , le coefficient $M_{i,j}^p$ est égal au nombre de chemins de longueur p de G dont l'origine est le sommet x_i et dont l'extrémité est le sommet x_j .

Correction exercice 10

On effectue une récurrence sur p . Pour $p = 1$ le résultat est immédiat car un chemin de longueur 1 est un arc du graphe. Le calcul de M^p , pour $p > 1$ donne :

$$M_{i,j}^p = \sum_{k=1}^{k=n} M_{i,k}^{p-1} \times M_{k,j}$$

Or tout chemin de longueur p entre x_i et x_j se décompose en un chemin de longueur $p - 1$ entre x_i et un certain x_k suivi d'un arc reliant x_k et x_j . Le résultat découle alors de l'hypothèse de récurrence suivant laquelle $M_{i,k}^{p-1}$ est le nombre de chemins de longueur $p - 1$ joignant x_i à x_k .

Fin correction exercice 10

Dans les exercices suivants, chaque site i , initialement, possède une valeur α_i et on veut au final que tous les sites aient la somme de toutes les valeurs ce trouvant sur chaque site initialement c'est à dire pour chaque site i on veut obtenir $\sum \alpha_i$.

Exercice 11 – Routage dans l'hypercube

On appelle hypercube de dimension n et on note $H(n)$, le graphe dont les sommets sont les mots de longueur n sur un alphabet à deux lettres 0 et 1 et dont deux sommets sont adjacents si et seulement si ils diffèrent en une seule coordonnée. Un sommet noté $x_1x_2\dots x_i\dots x_n$ est donc relié aux sommets $x_1x_2\dots \bar{x}_i\dots x_n$ avec $i = 1, 2, \dots, n$.

1. Représenter $H(3)$ et $H(4)$.
2. Montrer que $H(n)$ possède 2^n sommets.
3. Montrer que $H(n)$ est n -régulier.
4. Quel est le nombre d'arêtes ?
5. Quel est le diamètre de $H(n)$ où le diamètre est le maximum pris sur toutes les paires de sommets x et y de G , de la distance entre x et y .
6. Donner la matrice d'adjacence de l'hypercube $M_{H(3)}, M_{H(4)}$ et calculer $M_{H(3)}^k, M_{H(4)}^k$ avec $k \geq 2$.
7. Proposer un algorithme qui permettent de résoudre le problème initial et le faire tourner sur un exemple.

8. Donner la complexité en nombre de messages et en temps pour le modèle synchrone.

Correction exercice 11

1. facile
2. mot de longueur n sur un alphabet $\{0, 1\}$ implique qu'il y a $N = 2^n$ sommets.
3. On peut faire varier \bar{x}_i de n position. Il y a donc n voisins pour chaque sommet.
4. On sait $\sum_{i=1}^{2^n} d(x_i) = 2|E| = n2^n$ et donc $|E| = n2^{n-1}$.
5. le diamètre est donné par les sommets de coordonnées 00..0 et 11.11 est donc le diamètre est $n = \log_2 N$.
6. Sur un graphe d'hypercube, on applique l'algorithme suivant : on travaille par dimension. On envoie sa valeur à son voisin dans la même dimension. On fait la somme avec ce que l'on reçoit. Il faut envoyer après à une autre dimension avec un bit qui diffère. Ce bit doit être décalé afin d'envoyer à toutes les dimensions.
 - v <- valeur initiale
 - pour i allant de 1 à n
 - envoyer v aux voisins et v <- sommer toutes les valeurs reçues + v
 - fin pour
 - v contient la somme de toutes les valeurs
7. Nous avons $nN = n2^n$. Pour chaque sommet il faut envoyer à tous ses voisins. Le temps est n .
Une autre solution consiste à construire un anneau sur l'hypercube (coût 2^n), à faire circuler pas à pas la valeur des sites (coût 2^n) et avertir tout le monde (coût 2^n) donc un coût en nombre de messages de 3×2^n . En temps c'est 2^n .

Fin correction exercice 11

Exercice 12 – routage dans le graphe de De Bruijn

Le graphe de Bruijn $B(d, D)$ est le graphe orienté dont les sommets sont les mots de longueur D sur un alphabet de taille d , $d \geq 2$ et dont un sommet noté $x_1x_2\dots x_D$ est relié aux sommets $x_2\dots x_D\lambda$, λ étant une lettre quelconque de l'alphabet. On passe donc d'un sommet à un autre par un décalage à gauche avec adjonction d'une lettre.

1. Représenter le graphe de de Bruijn $B(2, 3)$.
2. Donner le nombre de sommets pour $B(d, D)$ et le nombre d'arcs.

- Donner le diamètre de $B(d, D)$.
- Donner la matrice d'adjacence M du graphe $B(2, 3)$. Calculer $M^k, k \geq 2$. Conclure.
- Proposer un algorithme qui permettent de résoudre le problème initial et le faire tourner sur un exemple.

Correction exercice 12

- La représentation est donné par la figure 5.

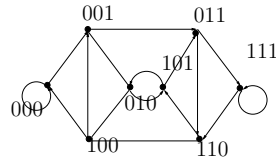


FIG. 5 – Un graphe de Bruijn $B(2, 3)$.

- Le nombre de sommet est d^D et d^{D+1} .
- Le diamètre est D . En effet, si on part d'un sommet donné après D décalages dans le même sens, on aura obtenu tous les sommets (donc le diamètre est au plus égal à D).
- Sur un graphe de de Bruijn $B(d, D)$, on applique l'algorithme suivant :
 - $v \leftarrow$ valeur initiale
 - pour i allant de 1 à D
 - envoyer v aux successeurs et $v \leftarrow$ sommer toutes les valeurs reçues
 - finpour
 - v contient la somme de toutes les valeurs

Fin correction exercice 12

Exercice 13 – Topologie quelconque Proposer un algorithme dans le cadre où la topologie du réseau est inconnue.

Correction exercice 13

Nous avons deux stratégies :

- je construis un arbre couvrant et j'applique un algorithme similaire de l'élection dans un arbre,
- ou je construis un anneau virtuel et j'applique l'algorithme simple qui consiste à envoyer sa valeur et et faire la somme de la valeur reçue avec la tienne.

Fin correction exercice 13

Exercice 14 – Liste des sommets à une distance 2 On définit le poids d'un sommet d'un réseau comme le nombre de sommets à distance inférieur ou égale à 2 de lui.

- Donner un algorithme distribué qui permette de calculer le poids d'un sommet x .
- Donner un algorithme qui indique à i la liste des voisins de ses voisins.

Correction exercice 14

Il est clair que la situation qui pose problème correspond à une clique de taille trois. Je pense que x doit envoyer un message vers ses voisins avec message où il y a deux champs dont un correspond au père c'est à dire x . Ensuite les voisins envoyer un message à ces voisins en indiquant le père du message c'est à dire x . C'est au sommet qui reçoit ce message de vérifier s'il n'a pas déjà reçu un message venant de x . Si c'est le cas il n'envoie pas d'accusé de réception. Si ce n'est pas le cas il envoie un accusé de réception a celui qui lui ea envoyé ce message. Donc les premiers voisins comptabilise le nombre d'accusé de réception pour les envoyer à x . De plus il faut faire attention au cas où $x \rightarrow y, x \rightarrow z, y \rightarrow u, z \rightarrow u$. Il ne faut que u renvoie deux accusé de réception à y et à z . Donc u doit également renvoyé un seul accusé de réception (pour cela mettre un variable booléenne pour régler cela).

voici le code : V_i voisin de i

$Voisindist2$ booléen.

Pour tout $y \in V_x$ faire
Envoyer ($Request(x)$) à y

Lors de la reception de $Request(x)$

Envoyer la liste de voisins ($Request(x, V_y - \{x\})$) à x

Lors de la reception de liste voisin $Request(x)$

Pour tout $y \in V_y$ faire
 $test := faux$;
Pour tout $z \in V_x$ faire
Si $z = y$ alors

```
FINSI      Si (test == faux) alors  
Voinsdist2 := Voinsdist2  $\cup$  {y}  
FINSI Finfaire
```

Fin correction exercice 14

Algorithme distribué
 TD – Séance n° 4

Exercice 1 – Terminaison

On se place dans le cadre d'un modèle atomique pour le comportement des processus c'est à dire dans lequel leurs temps de calcul sont considérés comme étant de durée nulle. Seules les communications prennent du temps. La réception d'un message par un site et l'envoi des messages qu'entraîne le traitement associé sont donc considérés comme seule action qui ne prend pas de temps. L'exécution d'un calcul peut être représentée par un diagramme de temps comme indiqué sur la figure 1 (Un point y représente un événement du type réception + émission). Dans ce modèle de comportement le problème de la terminaison revient donc à vérifier si tous les canaux sont simultanément vides.

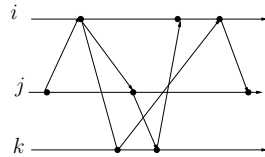


FIG. 1 – Une exécution dans le modèle atomique.

L'algorithme de Mattern Le principe consiste à compter le nombre message total E de messages émis, le nombre total R de messages reçus pour conclure à la terminaison lorsque ces deux nombres sont égaux.

Un observateur envoie sur le chacun des sites P_i une requête pour que ceux-ci indiquent les nombres E_i et R_i des messages respectivement émis et reçus depuis le début et attend les résultats. On dit ainsi qu'il a effectué une vague d'observation.

Soient $E = \sum_{i=1}^n E_i$ et $R = \sum_{i=1}^n R_i$ le nombre de message émis et reçus au total.

1. Dans le cas où $E = R$, est-ce que l'on peut conclure sur la terminaison ?
2. Un observateur fait deux vagues d'observation, ne lançant la seconde vague que lorsqu'il a obtenu tous les résultats de la première. Nous notons E' et R' les résultats de la première vague, E'' et R'' les résultats de la seconde. Nous notons par t un instant entre la fin de la première vague et le début de la seconde. Soient E^* et R^* le

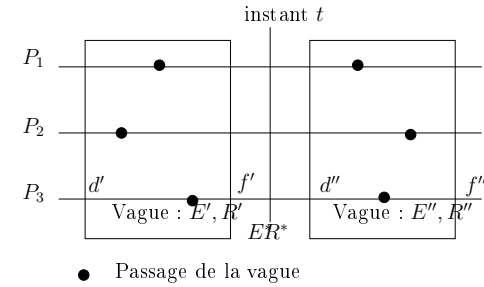


FIG. 2 – illustration d'une vague

nombre exacts de messages qui ont été émis et reçu avant t . Donner une relation entre R', R^*, E^* et E'' .

3. Montrer que si l'observateur constate que $E' = R''$ alors on peut conclure qu'aucun message n'était en transit à l'instant t .

Correction exercice 1

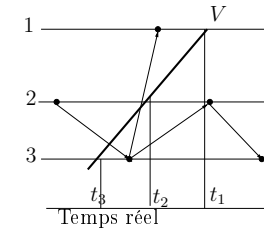


FIG. 3 – Une observation répartie

1. Malheureusement non, comme le montre le contre exemple suivant, figure 3 On a effet $E1 = R1$ avec : $e_1(t_1) = 0, e_2(t_2) = 1, e_3(t_3) = 0$ et $r_1(t_1) = 1, r_2(t_2) = 0, r_3(t_3) = 0$. La droite V sur le diagramme sépare le calcul observé en deux parties : pour chaque site il y a avant l'observation et après l'observation. Si les 3 sites étaient observés simultanément, i.e. $t_1 = t_2 = t_3$, on pourrait conclure à la terminaison à partir de l'égalité $E1 = R1$. Tout le problème de l'observation dans un contexte réparti provient

du fait que l'on ne peut garantir que les sites sont observés simultanément : il n'y a pas de référentiel de temps commun à l'ensemble des sites et les messages nécessaires pour réaliser l'observation ont des temps de transit arbitraires et a priori indépendants les uns des autres. Aucun site n'a connaissance de tous les instants réels t_1 , t_2 et t_3 , l'hypothèse de ce temps global ne peut donc servir qu'à l'analyse du problème et pas à l'algorithme qui le résoud !

2. Nous avons $R' \leq R^* \leq E^* \leq E''$
3. D'autre part, nous supposons qu'une réponse à une demande d'information ne quitte un site que lorsque ce site est au repos, et que la réponse envoyée tient compte de tous les envois et réception de message jusqu'à ce moment de repos. Nous supposons qu'après la deuxième vague, on ait $E'' = R'$ (nous sommes dans le cas où il n'y a aucun message reçu après t), on a alors aussi : $E'' = E^*$ (grâce à l'inégalité précédente); cela signifie qu'aucun site n'a envoyé de message entre l'instant t et le moment de sa réponse à la deuxième vague; notons S le premier site réactivé après le passage de la deuxième vague; S a été activé par la réception d'un message envoyé par un site S' ; l'envoi de ce message ne peut avoir lieu entre l'instant t et le passage de la seconde vague sur S' ni non plus avant l'instant t car alors le message aurait été en transit à l'instant t . S' (car nous sommes dans le cas de l'hypothèse, où il n'y a aucun message reçu après t) a donc envoyé son message après le passage de la seconde vague; il y a donc eu réactivation de S' après le passage de la seconde vague et avant la réactivation de S : d'où une contradiction sur le choix de S . En conséquence, aucun site ne sera réactivé après le passage de la seconde vague. De plus, s'il y a terminaison, deux successives donneront bien $R'' = E'$ lors de la seconde vague.

Remarque 1 Ce type de raisonnement est caractéristique du réparti. On cherche à conclure sur une propriété du système à un instant global t , la même sur tous les sites. Comme on ne peut garantir que tous les sites seront observés simultanément on calcule des approxiamtions des valeurs recherchées. La preuve consiste alors à montrer avec ces valeurs approchées qu'il a existé un instant t auquel la propriété recherchée était vérifiée. **Remarque 2** Si la terminaison n'est pas détectée par les vagues $V1$ et $V2$, il faut en lancer une autre $V3$ et on compare les valeurs collectées par $V2$ et $V3$ et ainsi de s suite. Dans tous les cas, i faut toujours deux vagues après la terminaison pour pouvoir la détecter. De plus l'algorithme ne donne aucune information, dans ce cas où il ne conclut pas à la terminaison, sur l'instant auquel il faudrait démarrer la prochaine vague afin d'avoir des chances de détecter la terminaison : le contrôleur C_α ne dispose d'aucune information sur les instants auxquels il doit lancer une vague.

Exercice 2 – Terminaison avec maintien d'un arbre On se place dans le contexte suivant : initialement tous les processus sont passifs à l'exception de l'un d'entre eux appelé P_0 . Un processus ne peut devenir actif que sur réception de messages, il peut alors envoyer à d'autres processus et les rendre donc éventuellement actifs; un processus ne peut émettre qu'un nombre fini de messages. De plus, pour simplifier nous supposons que P_0 ne peut qu'émettre des messages et ceci au début du calcul. Ce genre de communication et de calcul est appelé *calcul diffusant*. Il permet de modéliser un grand nombre de problèmes concrets dans lesquels P_0 joue le rôle de l'environnement qui déclenche l'activité d'un calcul dans le système.

Les outils sur lesquels s'articule la solution présentée sont ,une topologie de contrôle particulière (l'arbre) et que le fait que les délais de communication sont bornés par Δ .

Lorsqu'un processus est rendu actif par la réception d'un message il se place, s'il n'a pas de père, dans un arbre de contrôle avec l'émetteur du message comme père; il informe l'émetteur de cette relation parentale qui l'inclut dans ses fils. Lorsqu'un processus est passif et n'a pas de fils il sort de l'arbre en informant son père de la destruction de la relation parentale. Ainsi les processus actifs ou pères de processus actifs forment un arbre de racine P_0 (un processus n'existe qu'une fois au plus dans l'arbre). L'arbre de contrôle peut évoluer et se reconfigurer au cours du calcul en fonction de l'état actif ou passif dans lequel se trouve un processus à un instant donné et des messages échangés.

1. Le nombre de fils varie entre quelle valeurs?
2. Comment le site P_0 détecte la terminaison?
3. Comment le site P_i sait-il que le site P_j est son fils ou pas?
4. Donner les variables nécessaire à l'algorithme.
5. Donner l'algorithme.
6. Donner un algorithme voisin qui n'utilise pas le Δ .

1. Le nombre de fils varie de 0 à n .
2. Le principe de détection de la terminaison est alors le suivant : lorsque le processus P_0 n'a plus de fils, il n'y a plus d'activité dans le calcul sous-jacent : celui-ci est donc terminé.

3. Il reste un problème à régler : celui de la perception qu'a un processus du nombre de ses fils. Dans un contexte centralisé il n'y a pas de problème ; la perception de l'arbre se faisant en exclusion de ses modifications ; en répartit les transferts de messages consomment du temps. considérons un processus P_i qui n'a pas de fils dans l'arbre, qui envoie un message qui activera P_j et l'introduira dans l'arbre comme fils de P_i , après cet envoi P_i devenant passif. Le processus P_i ne doit pas alors conclure trop hâtivement qu'il n'a pas de fils dans l'arbre de contrôle : il doit attendre qu'un éventuel message de contrôle de P_j ait pu lui parvenir et l'informer d'une relation parentale. Si l'on considère que le temps de traitement sont nuls (il peuvent être absorbés dans Δ), P_i doit attendre 2Δ (un aller et retour de message) avant de conclure : il sait alors si P_j est un de ses fils ou non.

4. Chaque P_i se dote des variables suivantes :

- var $etat_i$: actif ou passif ;
- $nbfil_s_i$: $0 \dots n - 1$ initialisé à 0 ;
- $pere_i$: $1 \dots n \cup \{nil\}$ initialisé à nil ;
- $delecoule_i$ booléen initialisé à vrai ;

Le comportement du contrôle associé à P_i est le suivant. Les messages du type *controle* ont la valeur *sortir* ou *rentrer* dans l'arbre de contrôle.

5. Voici l'algorithme :

Lors de l'attente ou fin

$etat_i := passif$;

Lors de la reception de *message*(m, j)

si $pere_i := nil$; alors

$pere_i := j$

envoyer controle(*rentrer*) à P_j

$etat_i := actif$;

Lors du désir d'émettre *message*(m, i) vers j

on a alors $etat_i := actif$ et $pere_i \neq nil$;

envoyer *message*(m, i) à P_j

amerhorloge(2Δ) ;

$delecoule := faux$;

Lors du signal de l'horloge

$delecoule := vrai$;

Lors de la reception de controle(m)

cas $m = rentrer \rightarrow nbfil_s := nbfil_s + 1$

cas $m = sortir \rightarrow nbfil_s := nbfil_s - 1$

Lors de l'émission de controle(*sortir*)

possible seulement si $etat_i = passif$ et $nbfil_s_i = 0$ et $delecoule$;

$x := pere_i$;

envoyer controle(*sortir*) à P_x

$pere_i := nil$;

L'attente d'un délai de 2Δ permet à un processus de conclure à bon escient sur le nombre de ses fils : la perception qu'il en a lorsque il peut tester $nbfil_s$ et que celle-ci vaut 0 est alors exacte, (le booléen *delecoule* empêchant tout test de $nbfil_s$ lorsqu'il est à faux).

6. Une solution consiste : lorsque P_i envoie un message à P_j il l'inclut a priori dans ses fils ; à la reception du message, P_j lui répond de l'en sortir s'il est déjà dans l'arbre de contrôle (il a alors déjà un père). Ce principe est dual de celui exhibé dans l'algorithme présenté. L'inclusion d'un processus P_j dans l'arbre de contrôle, tel que le perçoit P_i l'émetteur du message, peut ainsi être faite par P_i :

- a posteriori après un délai de 2Δ après une émission. Si au terme de ce délai P_j n'a rien répondu à P_i , celui-ci n'a pas à le considérer comme un de ses fils. Il peut alors y avoir moins de messages de contrôle échangés que de messages de données, il y en a au pire 2 fois plus (un message *rentrer* et un message *sortir* par message de données échangé).

- a priori lors de l'émission. P_j peut alors répondre à P_i de la sortir de ses fils s'il est déjà dans l'arbre. Il n'y a alors que des messages de contrôle *sortir* et il y en aura le même nombre que de messages de données échangés par le calcul sous-jacent. De plus la variation $nbfil_s$ ne compte plus alors le nombre de fils potentiels de P_i mais le nombre de messages émis par P_i et non encore "acquittés" par *sortir*. (ce nombre est non borné a priori le processus P_i pouvant envoyer plusieurs messages à un même processus P_j sans savoir encore si P_j est ou non un de ses fils au moment considéré).

Algorithme distribué
TD – Séance n° 4

Exercice 1 – Perte d'un jeton sur un anneau unidirectionnel Supposons n sites placés sur l'anneau dans l'ordre $S_0, S_1, \dots, S_{n-1}, S_0$. Si le jeton se perd par exemple entre S_i et S_{i+1} deux choses sont à faire :

- détecter la perte du jeton
- en générer un nouvel exemplaire.

Si le jeton se perd et que plusieurs sites le détectent il est essentiel qu'un et un seul d'entre eux le régénère ; se pose donc le problème d'exclusion.

1. Expliquer pourquoi le fait d'avoir un mécanisme d'acquiescement du jeton entre deux sites ne fonctionnent pas. L'idée est de valuer le jeton et à utiliser cette valeur pour laisser sur chaque site une trace qui donnera les informations nécessaires pour détecter la perte éventuelle du jeton et le régénérer si tel est le cas avec la bonne valeur.
2. Appelons s_i la variable d'état locale à S_i dans laquelle celui-ci place la valeur v_j du jeton lorsqu'il le possède et cela avant de la transmettre à S_{i+1} . Comment détecter la perte d'un jeton ? Quel est le critère qu'un site devra vérifier lors de la perte d'un jeton ?
3. Que se passe-t-il pour le site S_0 ? Proposez une solution.
4. On peut de manière très générale supposer que les sites peuvent tomber en panne et qu'alors un protocole sous-jacent reconfigure l'anneau. Proposer une idée pour une reconfiguration simple.
5. Proposer maintenant l'algorithme.

Correction exercice 1

1. La première idée consiste à utiliser un simple mécanisme d'acquiescement du jeton entre deux sites. Mais c'est insuffisant pour résister à la perte de l'acquiescement. En effet la perte d'acquiescement peut entraîner la présence de plusieurs jetons. L'idée est de valuer le jeton et à utiliser cette valeur pour laisser sur chaque site une trace qui donnera les informations nécessaires pour détecter la perte éventuelle du jeton et le régénérer si tel est le cas avec la bonne valeur.

2. Appelons s_i la variable d'état locale à S_i dans laquelle celui-ci place la valeur v_j du jeton lorsqu'il le possède et cela avant de la transmettre à S_{i+1} . L'idée est la suivante : lorsque S_i passe le jeton à S_{i+1} il arme une horloge de garde locale hg_i à une valeur Δ ; lorsque ce délai est écoulé le site S_i va chercher à savoir si le jeton s'est perdu entre son prédécesseur S_{i-1} et lui-même, ce qui règle le problème d'exclusion précédent : si le jeton est perdu un seul site le détectera et donc le régénèrera. Il s'agit donc de trouver un test qui indique la perte du jeton entre S_{i-1} et S_i . Considérons à un instant la configuration suivante des variables d'état :

$$(s_0 = x, s_1 = x, \dots, s_{i-1} = x, s_i = x - 1, \dots, s_{n-1} = x - 1)$$

le jeton a fait $n - 1$ tours et le n ème tour jusqu'à S_{i-1} qui l'a envoyé à S_i . Si le jeton s'est perdu entre S_{i-1} et S_i ($i \neq 0$) on a donc $s_{i-1} \neq s_i$. Le protocole est donc simple : à l'échéance de son horloge de garde le site demande à son voisin de gauche (prédécesseur) la valeur de sa variable d'état : si celle-ci est égale à s_i le site S_i conclut que le jeton n'est pas perdu entre son prédécesseur et lui-même, dans le cas contraire il conclut à la perte et le régénère avec la valeur obtenue de son voisin (il faut que la demande-réponse utilise un protocole fiable).

3. L'anneau présente un point singulier S_0 : le début du tour. si le jeton est en transit ou se perd entre S_{n-1} et S_0 toutes les variables d'état sont égales ; le test de détection est donc pour S_0 l'égalité de s_0 avec la valeur de s_{n-1} .
4. On peut de manière très générale supposer que les sites peuvent tomber en panne et qu'alors un protocole sous-jacent reconfigure l'anneau. Tout site est alors doté de deux variables vg_i et vd_i qui définissent respectivement son voisin de gauche et son voisin de droite ; ce protocole garantissant que les sites sont toujours placés dans l'ordre de leurs identités sur l'anneau. Dans le cas S_0 est défaillant, c'est S_1 qui marquera alors le début du tour de l'anneau, etc. De manière générale le site début de tour, et qui doit donc incrémenter v_j , est le site tel que $i < vg_i$; grâce au placement ordonné des sites sur l'anneau ce site est unique. Le protocole de détection et de régénération est finalement le suivant pour un site S_i quelconque :

5. **visite du jeton attendre** jeton(v_j) de S_{vg_i}
désarmer hg_i ;
<utiliser le jeton conformément aux droits qu'il offre >
si $i < vg_i$ **alors** (début d'un nouveau tour) $v_j := v_j + 1$
fin
 $s_i := v_j$
envoyer jeton(v_j) à S_{vd_i}

$armer(hg_i, \Delta)$

Lors de l'échange ou S_i a besoin du jeton et ne l'a pasDébut

demander la valeur de s_{vg_i} , à S_{vg_i} ;

(S_{vg_i} ne répond que s'il n'a pas le jeton)

si $i < vg_i$ et $s_i = s_{vg_i}$ (S_i : début du tour)

ou $i > vg_i$ et $s_i \neq s_{vg_i}$ (S_i : banalisé)

alors (le jeton(s_{vg_i}) s'est perdu entre les deux sites)

créer un jeton(s_{vg_i}) (S_i est alors le jeton)

finsi

finsi

Fin correction exercice 1

Exercice 2 – Perte d'un jeton sur un anneau unidirectionnel (suite)

Supposons n sites placés sur l'anneau dans l'ordre $S_0, S_1, \dots, S_{n-1}, S_0$. Si le jeton se perd par exemple entre S_i et S_{i+1} deux choses sont à faire :

- détecter la perte du jeton
- en générer un nouvel exemplaire.

Si le jeton se perd et que plusieurs sites le détectent il est essentiel qu'un et un seul d'entre eux le régénère ; se pose donc le problème d'exclusion. Dans cet exercice nous proposons une alternative aux horloges de gardes : nous proposons d'utiliser deux jetons *ping* et *pong* ayant pour valuation respectivement nbi et nbo .

1. Pour détecter la perte d'un jeton nous allons utiliser un invariant. Proposez un invariant. Comment un jeton arrivant sur un site S_i peut détecter la perte de l'autre.
2. Donner l'algorithme de la reception des jetons *ping* et *pong* sur un site S_i . Donner également l'algorithme de reception des jetons sur un site S_i .
3. Quel est le défaut de l'algorithme. Modifier-le

Correction exercice 2

1. Afin d'éliminer les horloges de garde et le placement ordonné des sites sur l'anneau une idée simple consiste à introduire plusieurs jetons dont chacun est chargé de détecter la perte éventuelle des autres : tant qu'il y a un jeton le système peut donc fonctionner. Une étude probabilistique peut indiquer le nombre de jetons à utiliser, en fonction des caractéristiques des canaux, pour la probabilité qu'il y ait toujours au moins un jeton soit aussi proche de un que l'on veut. Nous nous limiterons à deux jetons.

2. **Le protocole** Soit *ping* et *pong* les deux jetons ; ils sont valués respectivement par nbi et nbo qui comptent le nombre de fois où ils se sont rencontrés et sont tels que $nbi + nbo = 0$. (Leurs valeurs initiales sont 1 et -1). Chaque site est doté d'une variable m_i dans laquelle le jeton dépose sa valeur (trace) lorsqu'il est sur ce site. Lorsqu'un jeton, par exemple *ping*(nbi), arrive sur S_i , si $m_i \neq nmi$, le jeton ou le site ont vu l'autre jeton depuis le précédent passage de *ping* sur S_i ; dans le cas contraire le site S_i peut conclure à la perte de *pong* et le régénérer. Comme on le voit ni les identités de sites, ni les horloges de garde sont nécessaires.

3. Le comportement d'un site quelconque S_i peut être décrit par :

Lors de la reception du jeton *ping*(nbi)

Si $m_i \neq nbi$

$m_i := nbi$

Sinon régénérer *pong*(nbo)

$nbi := nbi + 1$;

$nbo := -nbi$;

Lors de la reception du jeton *pong*(nbo)

(analogue au cas précédent en intervertissant les deux jetons)

Lors de la reception des deux jetons (sur un site)

$nbi := nbi + 1$;

$nbo := nbo - 1$;

Ce protocole revient à maintenir invariante la relation $nbi + nbo = 0$. Les variables manipulées peuvent être bornées. En effet, les deux jetons se rencontrent au plus une fois par site et par tour d'anneau, soit au plus n fois par tour d'anneau : les variables m_i des sites ne peuvent donc prendre au plus que n valeurs différentes (en valeurs absolues). Les compteurs nbi et nbo peuvent donc être utilisés modulo $n + 1$: entre deux visites à S_i un compteur n'a pu être incrémenté que de n au plus (en valeur absolue). D'autres mises en oeuvre de ce même principes sont possibles.

4. Dans la précédente un tour de jeton est nécessaire pour détecter la perte de l'autre ; on peut vouloir la détecter "au plus tôt" : si *pong* s'est perdu entre S_i et son successeur sur l'anneau S_j alors *ping* détecte la perte en S_j et non pas après un tour supplémentaire d'anneau. Pour réaliser cela les jetons doivent être valués non avec le nombre de leurs rencontres mais avec le nombre de sites visités qu'ils laissent, en trace, dans ces sites. Ainsi *ping* peut constater que *pong* est passé en S_i et qu'il ne l'a pas rencontré et qu'il n'est pas arrivé en S_j .

Fin correction exercice 2

Exercice 3 – Extension de l’algorithme de Naimi-Tréhel

L’algorithme de Naimi-Tréhel a pour but de gérer de façon distribuée, l’accès à une ressource critique, d’un ensemble de noeud (ou site). L’algorithme, à la manière du Token Ring, va utiliser un jeton pour symboliser l’autorisation d’accès à une ressource critique. Cet algorithme gère les noeuds du réseau en les organisant sous forme de deux structures. Tout d’abord un arbre va gérer la transmission des demandes à travers le réseau. Ensuite une file d’attente va stocker l’ordre de ces demandes et ainsi permettre la transmission du jeton d’un noeud à l’autre. Chaque noeud doit donc connaître le noeud situé juste au dessus de lui dans l’arborescence (appelé last), et le noeud qui lui succède dans la file d’attente (appelé next). Les noeuds du réseau qui veulent utiliser la ressource critique, vont utiliser un message type appelé *TOKENREQUEST* afin de rentrer dans la file d’attente. Ainsi, lorsqu’un noeud de l’arbre a besoin de la ressource, il envoie le message *TOKENREQUEST* à son last. A partir de là, le noeud qui reçoit la demande n’a que 2 options : - Si il n’est pas racine de l’arbre, il transmet la demande à son last puis modifie cette valeur de last pour pointer vers le demandeur. - Si il est racine de l’arbre, il est donc le destinataire du message. Il modifie la valeur de son next pour pouvoir lui transmettre le jeton dès sa sortie de la section critique. Il en fait de même avec son last pour pointer vers le demandeur.

Un invariant de cet algorithme est que le noeud racine de l’arbre sera également le dernier noeud de la file d’attente.

Cependant, une panne d’un noeud dans le réseau peut provoquer la perte irrémédiable d’information. En particulier, la perte du jeton peut survenir, si le noeud en panne est en possession du jeton, ou si il est le successeur dans la file d’attente du possesseur du jeton. Cette erreur pourrait compromettre l’intégrité du système car aucun des noeuds du réseau ne pourrait accéder à la ressource critique.

Les trois types de pannes que l’algorithme de Naimi-Tréhel doit surporter :

1. Le sommet qui tombe en panne appartient à l’arbre.
2. Le sommet qui tombe en panne appartient à la file.
3. Le sommet qui tombe en panne possède le jeton.

1. Quel est le mécanisme peut-on introduire afin de garantir un minimum de tolérance aux pannes ?
2. Nous proposons plusieurs mécanismes selon le type de panne. Soit S_i le site qui détecte la panne :

- (a) Le mécanisme $M1$: il apparaît quand le sommet S_i a reçu le message $\langle COMMIT \rangle$ et le nombre de sites en panne est inférieur à k .

- (b) Le mécanisme $M2$: il apparaît quand le sommet S_i a reçu le message $\langle COMMIT \rangle$ et le nombre de sites en panne est supérieur ou égal à k .
- (c) Le mécanisme $M3a$: il apparaît quand S_i n’a pas reçu de message $\langle COMMIT \rangle$ et est le seul qui a détecté une panne.
- (d) Le mécanisme $M3b$: il apparaît quand S_i n’a pas reçu de $\langle COMMIT \rangle$ et plusieurs sites ont détecté une panne.

3. Comment le site S_i peut détecter une panne pour chaque type de mécanisme ?
4. Donner les algorithmes associés à chaque mécanisme et préciser leur complexité.
5. Comment reconstruire l’arbre logique pour le mécanisme $M3$?
6. Comment gérer le fait que la position augmente tout le temps ?

Correction exercice 3

1. Afin de tolérer les pannes, l’extension proposée inclut un mécanisme avec accusé de réception. Ce mécanisme est employé lors de chaque demande d’accès à la ressource. Cet accusé de réception, appelé $\langle COMMIT \rangle$, est composé de deux valeurs :

- la position du receveur dans la file d’attente ;
- un tableau contenant les k prédécesseurs de l’expéditeur du $\langle COMMIT \rangle$ (k étant un élément paramétrable)

Lorsqu’une panne est détectée par un site S_i , ce dernier commence la procédure de réparation. La dite procédure comporte 4 mécanismes, chacun étant exécutée suivant le contexte de la panne.

2. Une panne est détectée grâce aux messages $\langle COMMIT \rangle$. Lors d’une demande de jeton, un site S_i attend un message $\langle COMMIT \rangle$ de son père S_j . Deux cas sont alors possibles :

- Soit il reçoit, auquel cas il doit maintenant vérifier régulièrement la vie de S_j (également prédécesseur dans la file d’attente). Cette vérification est réalisée au moyen de messages d’écho vers S_j ; on demande s’il est en vie et il doit répondre favorablement sinon on applique le mécanisme $M1$.
- Soit il ne reçoit pas de message $\langle COMMIT \rangle$ dans un délai fixé après sa demande. Dans ce cas, on applique le mécanisme $M3$.

Une autre panne peut-être détectée grâce aux messages $\langle AREYOUALIVE \rangle$.

Pour tout $S_j \in$ aux k prédécesseurs
envoyer($\langle ARE_YOU_ALIVE \rangle$) à S_j

mécanisme	contexte
M1 (page ??)	S_i a reçu le message <COMMIT> et le nombre de sites en panne est inférieur à k
M2 (page ??)	S_i a reçu le message <COMMIT> et le nombre de sites en panne est supérieur ou égal à k
M3-a (page ??)	S_i n'a pas reçu de message <COMMIT> et est le seul qui a détecté une panne
M3-b (page ??)	S_i n'a pas reçu de message <COMMIT> et plusieurs sites ont détecté une panne

TAB. 1 – Origine des pannes et mécanismes associés

```

attendre( 2Tmsg )
  Si réception( <I_AM_ALIVE> ) alors
    sortir de la boucle Pour
  Sinon Supprimer ( $S_j$ ) du tableau des prédécesseurs

```

Algorithme exécuté lors de la réception du message <ARE_YOU_ALIVE> :

```

next $_j$  :=  $S_i$ 
envoyer( <I_AM_ALIVE> ) à  $S_i$ 

```

Soit S_k un site recevant un message <AREYOUALIVE> en provenance de S_i, S_k renverra <IAMALIVE> systématiquement à S_i , et ce même s'il n'est plus dans la file. En conséquence, S_i n'aura le jeton que si S_k le demande et l'arbre logique ne sera plus fonctionnel. On affecte la valeur -1 aux sites qui ne sont pas ou plus dans la file d'attente et on exécute les instructions suivantes lors de la réception d'un message <AREYOUALIVE>

```

Si ( $position_k \neq 1$ ) alors
  next $_k$  :=  $S_i$ 
  Envoyer(<IAMALIVE>) à  $S_i$ ;

```

- Le mécanisme M2. On se préoccupe ici du cas où <IAMALIVE> n'a pas été reçu durant le mécanisme M1. Dans ce cas, on applique l'algorithme suivant qui consiste d'abord à expédier un message <SEARCHPREV> à tout le monde. Ensuite, pendant un délai fixé (ici à $2Tmsg$) on va recueillir les réponses. A noter que seuls les sites précédents dans la file vont répondre (algorithme 5). ON sélectionne ensuite, parmi

les réponses, le site qui se trouve le plus près dans la file (algorithme 6). Une fois le premier prédécesseur trouvé, on tente de se connecter à celui-ci.

Lors de la réception du message <SEARCHPREV, position $_i$ > en provenance du site S_i par le site S_j , on exécute l'algorithme suivant traduisant le mécanisme M2

```

site $_{etu}$  := null
position $_{etu}$  := null
diffuser( <SEARCH_PREV, position $_i$ > )
attendre( 2Tmsg )
Si site $_{etu} \neq$  null alors
  (Je n'est reçu aucun message ACK_SEARCH_PREV)
  générerJeton()
  position $_i$  := 0
  entrerEnSectionCritique()
  Sinon envoyer( <CONNECTION> ) à site $_{etu}$ 
  position $_i$  := position $_{etu}$  + 1

```

Lors de la réception du message <SEARCH_PREV, position $_i$ > en provenance du site S_i par le site S_j , on exécute l'algorithme suivant :

```

Si (( $position_k \neq -1$ ) et ( $position_j < position_i$ )) alors
  Envoyer(<ACKSEARCHPREV, position $_j$ >) à  $S_i$ ;

```

Lors de la réception du message <ACKSEARCHPREV, position $_i$ > du site S_i

```

Si ( $position_{etu} < position_i$ ) alors
  site $_{etu}$  :=  $S_i$ 
  position $_{etu}$  := position $_i$ 

```

Lors de la réception du message <CONNECTION> du site S_i par S_j , ce dernier exécute l'algorithme suivant (algorithme servant à connecter le site S_j à la file d'attente) :

```

next $_j$  :=  $S_i$ 

```

Le mécanisme M2 n'est appliqué que lorsque tous les prédécesseurs de la table ont été éliminés, faute de réponse à <AREYOUALIVE>. IL faut par conséquent mettre à jour cette table. Trivialement, il suffit de récupérer le tableau du $site_{etu}$ (s'il n'est pas nul).

Pour cela deux possibilités :

- soit le $site_{elu}$ envoie le tableau dans le message $\langle ACKSEARCHPREV \rangle$;
- soit le $site_{elu}$ envoie le tableau dans le $\langle COMMIT \rangle$ résultant de la demande de connexion.

Nous choisissons la seconde car, dans la première, le tableau serait transmis systématiquement ce qui n'est pas nécessaire.

4. Le mécanisme $M3$ Nous avons vu comment traiter le cas où le message $\langle COMMIT \rangle$ avait bien été reçu. Dans le cas contraire c'est $M3$ qui est chargé de traiter la défaillance. Cependant, il peut arriver que plusieurs sites détectent la panne. C'est pourquoi, il existe deux variantes de $M3$: $M3-a$ et $M3-b$.

(a) Le mécanisme $M3a$: Nous sommes ici dans le cas où un seul site S_i a détecté une panne.

Algorithme traduisant le mécanisme $M3 - a$

```

 $site_{elu} := null$ 
 $position_{elu} := null$ 
 $site_{elu\_avoir\_next} := null$ 
diffuser(  $\langle SEARCH\_QUEUE \rangle$  )
attendre(  $2Tmsg$  )
Si  $site_{elu} == null$  alors
(Je n'est reçu aucun message  $ACK\_SEARCH\_QUEUE$ )
    générerJeton()
     $position_i := 0$ 
    Sinon Si  $site_{elu\_avoir\_next} == FAUX$  alors
        envoyer(  $\langle REQUEST \rangle$  ) à  $site_{elu}$ 
        Sinon envoyer(  $\langle CONNECTION \rangle$  ) à  $site_{elu}$ 
         $position_i := position_{elu} + 1$ 

```

Lors de la réception du message $\langle SEARCH_QUEUE \rangle$ de S_j par le site S_i :

```

Si  $position_i \neq -1$  alors
    Si  $next_i \neq null$  alors
        envoyer(  $\langle ACK\_SEARCH\_QUEUE, position_i, VRAI \rangle$  ) à  $S_j$ 
    Sinon envoyer(  $\langle ACK\_SEARCH\_QUEUE, position_i, FAUX \rangle$  ) à  $S_j$ 

```

Lors de la réception du message $\langle ACK_SEARCH_QUEUE, position_j, AVOIR_NEXT \rangle$ de S_j par le site S_i :

```

Si  $position_j > position_{elu}$  alors
     $site_{elu} := S_j$ 

```

```

 $position_{elu} := position_j$ 
 $site_{elu\_avoir\_next} := AVOIR\_NEXT$ 

```

(b) Le mécanisme $M3b$:

Cette variante concerne le cas où plusieurs sites détectent une panne.

En effet, si plusieurs sites utilisent le mécanisme $M3 - a$ simultanément, plusieurs jetons sont générés ; En conséquence, la file d'attente devient inconsistante et le principe de l'unicité du jeton n'est plus valable.

Dans ce mécanisme, un site est candidat pour la reconstitution de la file d'attente à partir du moment où il diffuse le message $\langle SEARCH_QUEUE \rangle$. Un site est considéré comme élu s'il est toujours candidat après $2Tmsg$.

L'algorithme utilisé lors de la réception d'un message $\langle SEARCH_QUEUE \rangle$ est le suivant traduisant le mécanisme $M3 - b$:

```

 $site_{elu} := moi\_même$ 
 $nbr\_entree\_en\_sc_{elu} := X$ 
 $site_{vivant} := null$ 
 $site_{vivant\_avoir\_next} := null$ 
 $position_{vivant} := null$ 
diffuser(  $\langle SEARCH\_QUEUE \rangle$  )
attendre(  $2Tmsg$  )
Si  $site_{elu} == moi\_même$  alors
(je suis toujours candidat)
    Si  $site_{vivant} == null$  alors
(aucun  $ACK\_SEARCH\_QUEUE$  reçu)
        générerJeton()
         $position_i := 0$ 
        Sinon Si  $site_{vivant\_avoir\_next} == FAUX$ 
            envoyer(  $\langle REQUEST \rangle$  ) à  $site_{vivant}$ 
            Sinon envoyer(  $\langle CONNECTION \rangle$  ) à  $site_{vivant}$ 
            et  $position_i := position_{vivant} + 1$ 
        (un autre site est élu)
         $pere_i := site_{elu}$ 
        envoyer(  $\langle REQUEST \rangle$  ) à  $site_{elu}$ 

```

Une conséquence de ce changement est la modification de l'algorithme employé lors de la réception du message $\langle SEARCH_QUEUE \rangle$ de S_j par le site S_i :

C'est l'algorithme au message $\langle SEARCH_QUEUE \rangle$

```

(j'ai une position dans la file d'attente)
Si  $position_i \neq -1$  alors
  Si  $next_i \neq null$ 
    envoyer( <ACK_SEARCH_QUEUE,  $position_i$ , VRAI> ) à  $S_j$ 
  Sinon envoyer( <ACK_SEARCH_QUEUE,  $position_i$ , FAUX> ) à  $S_j$ 
Sinon (je n'ai pas de position)
  Si  $attente\_jeton == VRAI$  alors
     $site_{elu} := \text{determinerLeGangant}( site_{elu}, S_j )$ 
     $nbr\_entree\_en\_sc_{elu} := site_{elu}.nbr\_entree\_en\_sc$ 
     $pere_i := next_i$ 

```

- (c) Reconstruction de l'arbre A la fin des mécanismes M3, la file d'attente est reconstituée mais pas l'arbre logique. Le problème est de savoir à quel moment un site i considère qu'un autre site j est l'élu? c'est à dire considérer le site j comme un père. En fait, les sites reçoivent autant de messages <SEARCH_QUEUE> qu'il y a de sites qui ont détecté une panne. La difficulté réside alors dans le fait qu'aucun site ne sait à quel moment il ne recevra plus de message <SEARCH_QUEUE> et donc comment connaître l'élu.

Pour remédier à celà, nous utilisons un minuteur lancé à la réception du premier message <SEARCH_QUEUE>. Ce minuteur s'arrête au bout de $2T_{msg}$ et l'élection est effectuée par rapport à l'ensemble des réponses reçues. Le site élu est stocké dans la variable $site_{elu}$ et il suffit de mettre à jour la variable $pere$.

- (d) Position : incrémentée vers l'infini La position augmente à chaque demande de jeton mais ne diminue jamais. Arrivée à une certaine valeur ceci peut causer une erreur de segmentation.

Solution envisagée Afin de résoudre ce problème, nous avons mis en place la stratégie suivante :

```

- choisir une valeur critique ;
- lors de la réception du jeton par un site  $S_i$  (algorithme Réception du jeton et mise à jour des positions :
  Si ( $next_i \neq null$ ) et ( $position_i \geq \text{VALEUR\_CRITIQUE}$ ) alors
    envoyer( <UPDATE_POSITION, 0> ) à  $next_i$ 
  entrerEnSectionCritique()
  Si  $next_i \neq null$  alors
    envoyer( <JETON> ) à  $next_i$ 
     $next_i := null$ 
   $position_i := -1$ 

```

```

- lors de la réception du message <UPDATE_POSITION, position> par  $S_i$  (pour la Mise à jour de la position )
   $position_i := position + 1$ 
  Si  $next_i \neq nul$  alors
    envoyer( <UPDATE_POSITION,  $position_i$ > ) à  $next_i$ 

```

Fin correction exercice 3

Algorithme distribué
TD – Séance n° 6

$$P(x) = a_{t-1}x^{t-1} + a_{t-2}x^{t-2} + \dots + a_1x + s$$

Les ombres sont calculées par $s_i = P(i) \pmod p$.

1. Application à $n = 12, t = 3, s_1 = 4, s_5 = 9, s_6 = 12$. Quel est le secret ?

Exercice 1 – Généraux byzantins

On considère quatre processus répartis chacun sur un site dont au plus un est byzantin, et le protocole d'accord suivant : lors de la première phase, chaque processus diffuse sa valeur initiale à tous les autres, attend d'avoir obtenu eux réponses (le réseau est asynchrone et les messages signés) et choisit la valeur majoritaire parmi les trois valeurs (la sienne et les deux reçues, en cas d'égalité il choisit la plus petite. Les phases 2 et 3 sont identiques en partant de la valeur choisie à la phase précédente. La valeur de l'accord est la valeur choisie à la fin de la troisième phase.

1. Ce protocole est-il une solution correcte au problème de l'accord ?

Correction exercice 1

1. La réponse est non. Prenons un K_4 avec un félon. Le byzantin envoie une fois 1 et deux fois 2 à ces voisins. Le problème est lié au fait qu'un général félon ait pu envoyer deux valeurs différentes à deux généraux loyaux.

Fin correction exercice 1

Exercice 2 – Calcul d'un secret

Soit s un entier secret, n le nombre de sites et t une borne supérieure du nombre de félons. On se propose de construire n entiers, appelés les ombres du secret s , qui seront distribués entre les participants de telle sorte que :

- si au moins t participants se communiquent leurs ombres, alors ils peuvent reconstruire le secret,
- tout groupe de moins de t participants ne peut reconstruire s .

La méthode : choisir p un nombre premier plus grand que n et s . Par la suite tous les nombres sont calculés modulo p . Puis choisir $t - 1$ entiers $a_1, a_2, a_3, \dots, a_{t-1}$ et on construit le polynôme en la variable x :

Algorithmes distribués
TD – Séance n° 3

Exercice 1 – Détermination d'un centre

Considérons une grille $q \times q$, $q = \sqrt{n}$. Soit α le sommet au coin Nord-ouest.

1. Donner l'arbre de recouvrement $ART(\alpha)$.
2. Donner le centre $C(ART(\alpha))$.
3. Calculer $R(G)$ et $R(ART(\alpha))$.

Exercice 2 – Borne supérieure atteinte pour le calcul d'un centre

1. Donner un graphe pour lequel le majorant sur la complexité pour le calcul d'un centre est atteint.

Exercice 3 – Le cas du modèle asynchrone pour la recherche d'un centre et d'une médiane

- 1.

Exercice 4 – Parcours en profondeur

Dans cet algorithme nous supposons que chaque site connaît l'identité des ses voisins.

1. Donner un algorithme de parcours en profondeur qui construit un arbre.
2. Evaluer sa complexité en nombres de messages.
3. Evaluer sa complexité en temps.
4. Evaluer sa complexité en nombre de bits.