1 Horloges

Principe : L'horloge logique entière (H) date toute action avec une valeur entière supérieure à celle de toutes les actions qui la précèdent au sens de la causalité. Une valeur de date logique est attribuée à chaque action. L'horloge logique entière respecte les liens de causalités, ne permet pas d'obtenir une unique observation car l'ordre qu'elle induit sur les actions n'est pas total (on pourra intervertir les actions indépendantes comme on le souhaite pour construire des observations valides différentes). L'ordre construit par H est donc un ordre partiel puisque certaines actions (les actions indépendantes) ne sont pas comparables selon la relation d'ordre induite.

Horloge entière, site S_i

1 initialisation
2 $h_i \leftarrow 1$
3 action interne
4 $h_i \leftarrow h_i + 1$
5 action d'émission
6 $h_i \leftarrow h_i + 1$
7 **envoyer** (h_i ...)
8 action de réception
9 **recevoir** (h ...)
10 $h_i \leftarrow max(h, h_i) + 1$

Principe : Alors de les horloges logiques entières H et la fonction d'estampillage K définissent chacune un ordre — partiel pour H (actions indep. non comparables) et total pour K (Même les actions indépendantes sont comparables ; mais, on pert l'information de leur indépendance) — sur les actions du système, de telle sorte que ces ordres sont compatibles avec la relation de causalité, l'horloge vectorielle répond à un autre besoin que les estampilles pusiqu'elle permet de conserver l'information que 2 actions sont indépendances. L'idée de base sous-jacente à cette horloge consiste à utiliser le passé des actions pour les dater. Inconvénients : Si beaucoup de sites => grosse horloges => messages "lourds"

Horloge vectorielle, site S_i

1 initialisation
2 $V_i \leftarrow \mathbb{1}_i$
3 action interne
4 $V_i \leftarrow V_i + \mathbb{1}_i$
5 action d'émission
6 $V_i \leftarrow V_i + \mathbb{1}_i$
7 **envoyer** (V_i ...)
8 action de réception
9 **recevoir** (V ...)
10 $V_i \leftarrow MAX(V, V_i) + \mathbb{1}_i$

2 Instantanés

Principe : Envoyer un marqueur (message spécial) aux sites pour les avertir de faire leur sauvegarde locale.

Hypothèses :

— Canaux de communication FIFO (pour s'assurer que l'on a pas de messages post-pré)

— Utilisation d'un marqueur pour séparer les messages envoyés en préclic des messages envoyés en postclic

— Un site initiateur (le début de l'execution de l'algo se fait sur ordre de l'app. de base)

Instantané avec marqueur, site S_i

```
   /* Canaux FIFO, utilise un marqueur                              */
 1 initialisation
 2     couleur_i ← blanc
 3 début de l'instantané
 4     couleur_i ← rouge
 5     envoyer ( [marqueur] ) aux voisins
 6 réception d'un marqueur
 7     si couleur_i == blanc alors
 8         couleur_i ← rouge
 9         envoyer ( [marqueur] ) aux voisins
10     fin si
```

Principe : L'application de contrôle ajoute une donnée (couleur) aux messages échangés par les applications de base.

Hypothèses :

— Canaux de communication NON FIFO

— Les applications de base doivent générées suffisament de messages pour que la demande de sauvegarde se fasse bien au travers du réseau (si un site ne génère jamais de messages, ces voisins ne recevront jamais la demande de sauvegarde)

Instantané avec lestage, site S_i

```
   /* Canaux non FIFO, algorithme de contrôle                       */
 1 initialisation
 2     couleur_i ← blanc
 3 début de l'instantané
 4     couleur_i ← rouge                          /* sur un seul site */
 5 réception d'un message m de l'app. de base
 6     recevoir (m à destination de S_j)
 7     envoyer (m, couleur_i) à S_j
 8 réception d'un message de type m
 9     recevoir (m, c)
10     si c == rouge et couleur_i == blanc alors
11         couleur_i ← rouge
12     fin si
13     traiter le message m
```

Instantané avec reconstitution de configuration, site S_i

```
   /* Réseau connexe non FIFO de N sites avec anneau de contrôle
   orienté                                                        */
 1 initialisation
 2     couleur_i ← blanc
 3     initiateur_i ← faux
 4     bilan_i ← 0
 5     EG_i ← ∅
       /* utilisé sur l'initiateur seulement :            */
 6     NbEtatsAttendus_i ← 0
 7     NbMsgAttendus_i ← 0
 8 début de l'instantané                          /* sur un seul site */
 9     couleur_i ← rouge
10     initiateur_i ← vrai
11     EG_i ← état local courant
12     NbEtatsAttendus_i ← N − 1
13     NbMsgAttendus_i ← bilan_i
14 réception d'un message m de l'app. de base
15     recevoir (m à destination de S_j)
16     envoyer (m, couleur_i) à S_j
17     bilan_i ← bilan_i + 1
18 réception d'un message de type m
19     recevoir (m, c)
20     si c == rouge et couleur_i == blanc alors
21         couleur_i ← rouge
22         EG_i ← état local courant
23         envoyer ( [état] EG, bilan_i ) sur l'anneau
24     fin si
25     si c == blanc et couleur_i == rouge alors     /* message prépost */
26         envoyer ( [msg] m ) sur l'anneau
27     fin si
28     bilan_i ← bilan_i − 1
29     traiter le message m
30 réception d'un message de type [état]  /* réception état local et bilan */
31     recevoir ( [état] EG, bilan )
32     si initiateur_i == vrai alors
33         EG_i ← EG_i ∪ EG
34         NbEtatsAttendus_i ← NbEtatsAttendus_i − 1
35         NbMsgAttendus_i ← NbMsgAttendus_i + bilan
36         si NbEtatsAttendus_i == 0 et NbMsgAttendus_i == 0 alors
37             FIN
38         fin si
39     sinon
40         envoyer ( [état] EG, bilan ) sur l'anneau
41     fin si
42 réception d'un message de type [msg]      /* réception message prépost
   retransmis */
43     recevoir ( [msg] m )
44     si initiateur_i == vrai alors
45         NbMsgAttendus_i ← NbMsgAttendus_i − 1
46         EG_i ← EG_i ∪ m
47         si NbEtatsAttendus_i == 0 et NbMsgAttendus_i == 0 alors
48             FIN
49         fin si
50     sinon
51         envoyer ( [msg] m ) sur l'anneau
52     fin si
```

Principe : Appliquer des modifications successives à l'algorithme de lestage pour pouvoir rassurer les sauvegardes locales aux sites sur le site initiateur de la sauvegarde. Cet algorithme doit : collecter les différents états locaux + collecter les messages pré-post + inclure une condition de terminaison de l'algorithme portant sur le nombre d'états reçus + les sites doivent savoir combien de messages préposts sont attendus.

Hypothèses :

- Réseau connexe
- N sites dans le réseau ET linitiateur le sait
- Canaux de communication non FIFO
- Anneau de contrôle orienté et passant par tous les sites (circuit hamiltonien) -> variable $succ_i$ sur chaque site ET cet anneau est déja établi.

3 vagues/demi-vagues/ondes

Principe : Une demi-vague ne précise pas vers quels voisins elle doit progresser : la demi-vague ne progressera que vers une sélection de voisins. Cependant, sans connaissance de la topologie sous-jacente, il est préférable de propager la demi-vague vers tous les voisins pour s'assurer que tous les sites du réseau soient atteints. Note, il n'y a pas de variables *parent*. On s'occupe juste de propager la demi vague à ses voisins (tous ses voisins pour être sûr que tous les sites soient atteints dans le cas où on ne connaît pas la topologie). Ici, on veut juste atteindre tous les sites.

Demi-vague générique

```
1  initialisation
2      couleur_i ← blanc
3  début de la demi-vague                          /* sur l'initiateur */
4      couleur_i ← bleu                          /* action de demi-vague */
5      sélection des voisins prévenus
6      envoyer ( [bleu] -) à ces voisins
7  réception d'un message bleu
8      recevoir ( [bleu] -) de S_j
9      si couleur_i == blanc alors
10         couleur_i ← bleu                      /* action de demi-vague */
11         sélection des voisins prévenus
12         envoyer ( [bleu] -) à ces voisins
13     fin si
```

Principe : Introduction d'une variable *parent* pour retenir son site père et ainsi construire une arborescence. Ici, on fait l'algo de demi vague classique tout en changeant la variable couleur en une variable *parent*. On veut atteindre tous les noeuds, et, ce que l'on souhaite retenir comme valeur c'est la valeur de son père. C'est exactement le même algorithme !

Arborescence couvrante

```
1  initialisation
2      parent_i ← 0
3  début de la diffusion                           /* sur l'initiateur */
4      parent_i ← i                              /* action de demi-vague */
5      envoyer ( [bleu] -) à tous les voisins
6  réception d'un message bleu
7      recevoir ( [bleu] -) de S_j
8      si parent_i == 0 alors
9          parent_i ← j                          /* action de demi-vague */
10         envoyer ( [bleu] -) à tous les voisins
11     fin si
```

Principe : Une diffusion détermine une arborescence, dite arborescence de diffusion. La diffusion avec retour consiste à diffuser une valeur, puis à remonter l'information « diffusion terminée » depuis les feuilles de cette arborescence jusqu'à sa racine, c'est-à-dire l'initiateur. Un site attend donc autant de messages de retour qu'il a de fils dans l'arborescence.

Notes :

— Ici on utilise le fait que l'on ai construit une arborescence couvrante lors de la diffusion l'aller, pour l'utiliser pour les messages de retour. On doit donc utiliser la variable *parent*.

— La variable "$NbVoisinsAttendus$" sert a savoir lorsqu'un site a reçu tous les messages de retour, et donc, lorsqu'il peut envoyer son message de retour a son parent.

— La définition de la vague n'impose pas aux messages de retour de suivre le cheminement inverse des messages aller. On pourrait donc définir des algorithmes de vague sur des réseaux dont les communications ne sont pas bidirectionnelles (La seule condition c'est que le site initiateur soit le decideur -> le retour se fait vers l'initiateur)

— Comme pour la demi-vague, la définition d'une vague n'impose pas de diffuser les messages bleus à tous les voisins. Cependant, le choix des voisins doit être tel que tous les descendants de l'initiateur sont atteints

Diffusion avec retour

```
1  initialisation
2      parent_i ← 0
3      NbVoisinsAttendus_i ← nb de voisins
4  début de la diffusion          /* sur l'initiateur (détenteur de l'info) */
5      parent_i ← i
6      envoyer ( [bleu] -) aux voisins
7  réception d'un message bleu
8      recevoir ( [bleu] -) de S_j
9      si parent_i == 0 alors
10         parent_i ← j
11         NbVoisinsAttendus_i ← NbVoisinsAttendus_i − 1
12         si NbVoisinsAttendus_i > 0 alors
13             envoyer ( [bleu] -) aux voisins sauf S_j
14         sinon
15             envoyer ( [rouge] -) à S_j
16         fin si
17     sinon
18         envoyer ( [rouge] ) à S_j
19     fin si
20 réception d'un message rouge
21     recevoir ( [rouge] -) de S_j
22     NbVoisinsAttendus_i ← NbVoisinsAttendus_i − 1
23     si NbVoisinsAttendus_i == 0 alors
24         si parent_i == i alors
25             FIN
26         sinon
27             envoyer ( [rouge] -) à S_{parent_i}
28         fin si
29     fin si
```

Parcours en profondeur

```
1  initialisation
2      parent_i ← 0
3      n_i ← 0
4      VoisinsNonAtteints_i ← ensemble des voisins
5  début de la demi-vague                          /* sur l'initiateur */
6      parent_i ← i
7      n_i ← 1
8      si ∃S_k ∈ VoisinsNonAtteints_i alors
9          envoyer ( [bleu] 2) à un voisin S_j
10         VoisinsNonAtteints_i ← VoisinsNonAtteints_i\{S_j}
11     sinon
12         FIN
13     fin si
14 réception d'un message bleu
15     recevoir ( [bleu] p) de S_j
16     si parent_i ≠ 0 alors
17         envoyer ( [rouge] p ) à S_j                 /* site déjà visité */
18     sinon
19         n_i ← p
20         parent_i ← j
21         VoisinsNonAtteints_i ← VoisinsNonAtteints_i\{S_j}
22         si ∃S_k ∈ VoisinsNonAtteints_i alors
23             envoyer ( [bleu] p + 1) à S_k
24             VoisinsNonAtteints_i ← VoisinsNonAtteints_i\{S_k}
25         sinon                          /* tous les voisins ont été visités */
26             envoyer ( [rouge] p + 1) à S_j
27         fin si
28     fin si
29 réception d'un message rouge
30     recevoir ( [rouge] p) de S_j
31     si ∃S_k ∈ VoisinsNonAtteints_i alors
32         envoyer ( [bleu] p ) à S_k
33         VoisinsNonAtteints_i ← VoisinsNonAtteints_i\{S_k}
34     sinon                              /* tous les voisins ont été visités */
35         si parent_i ≠ i alors
36             envoyer ( [rouge] p) à S_parent_i
37         sinon
38             FIN
39         fin si
40     fin si
```

Principe : Le calcul ne sera pas entièrement réalisé sur l'initiateur ; il sera réalisé partiellement sur chaque site recevant des messages. Le calcul global se décompose en calculs intermédiaires ne faisant intervenir que deux valeurs (la valeur courante du site, et celle reçue avec un message) + on ne peut prédire l'ordre dans lequel se feront les calculs intermédiaires. Le calcul d'une valeur dans un réseau peut être réalisé par une vague, basée sur l'algorithme de la diffusion avec retour

Hyptohèses :

— L'opérateur doit admettre certaines propriétés pour garantir un calcul correcte : binaire , associatif et commutatif. Si l'opérateur n'est pas idempotent => prendre des précautions supplémentaires.

Calcul d'une valeur par une vague avec l'opérateur $\odot$

```
1  initialisation
2      parent_i ← 0
3      NbVoisinsAttendus_i ← nb de voisins
4      valeur_i ← valeur quelconque, selon le calcul à effectuer
5  début du calcul                                   /* sur l'initiateur */
6      parent_i ← i
7      envoyer ( [bleu] -) aux voisins
8  réception d'un message bleu
9      recevoir ( [bleu] -) de S_j
10     si parent_i == 0 alors
11         parent_i ← j
12         NbVoisinsAttendus_i ← NbVoisinsAttendus_i − 1
13         si NbVoisinsAttendus_i == 0 alors      /* pas d'autre fils, retour
               direct */
14             envoyer ( [rouge] valeur_i) à S_j
15         sinon
16             envoyer ( [bleu] ) aux voisins sauf S_j
17         fin si
18     sinon                                  /* la vague a déjà été reçue */
19         envoyer ( [orange] ) à S_j
20     fin si
21 réception d'un message rouge
22     recevoir ( [rouge] k) de S_j
23     valeur_i ← valeur_i ⊙ k
24     NbVoisinsAttendus_i ← NbVoisinsAttendus_i − 1
25     si NbVoisinsAttendus_i == 0 alors
26         si parent_i == i alors
27             FIN
28         sinon
29             envoyer ( [rouge] valeur_i) à S_{parent_i}
30         fin si
31     fin si
32 réception d'un message orange                    /* pas de calcul */
33     recevoir ( [orange] ) de S_j
34     NbVoisinsAttendus_i ← NbVoisinsAttendus_i − 1
35     si NbVoisinsAttendus_i == 0 alors
36         si parent_i == i alors
37             FIN
38         sinon
39             envoyer ( [rouge] valeur_i) à S_{parent_i}
40         fin si
41     fin si
```

Principe : Les ondes correspondent à des explorations de réseaux qui, dans leur définition, sont plus générales que les vagues : l'onde remonte vers un site qui n'est pas obligatoirement l'initiateur. Les messages de retour ne suivent pas le cheminement inverse des messages aller, et les hypothèses sur le réseau sont plus faibles

3.1 Onde sur un anneau

Lorsque le réseau est un anneau (ou bien lorsqu'on dispose d'un anneau de contrôle dans un réseau quelconque), on peut mettre à profit cette connaissance de la topologie pour éviter d'utiliser une vague, qui oblige les messages à rebrousser chemin vers l'initiateur au lieu de poursuivre le tour de l'anneau.

3.2 Onde sur un arbre

Lorsque le réseau est un arbre (ou bien lorsqu'on dispose d'un arbre couvrant dans un réseau quelconque), on peut aisément parcourir le réseau (pour centraliser une information, calculer une valeur, etc.) en propageant l'exploration depuis les feuilles vers le «cœur» de l'arbre. Ce type d'exploration est une onde dont les initiateurs sont les feuilles, et dont au moins l'un des sites dans le réseau sera le décideur

Onde sur un arbre, 1ère version

```
1  initialisation
2      PèresPossibles_i ← ensemble des voisins
3      OndePartie_i ← faux
4  réception d'un message
5      recevoir ( – ) de S_j
6      PèresPossibles_i ← PèresPossibles_i \ {S_j}
7  | PèresPossibles_i | == 1 et OndePartie_i == faux
       /* Action d'onde                          */
8      envoyer ( – ) aux voisins dans PèresPossibles_i
9      OndePartie_i ← vrai
10 |PèresPossibles_i| == 0
       /* Fin de l'onde                          */
       /* Action de décision                        */
```

4 Elections

Principe : Les candidatures des sites sont envoyées sur l'anneau. Une liste de toutes les candidatures est faite. Une fois que cette liste est conçue, chaque site calcule le minimum de la liste => le minimum = ID du site élu.

Hypothèses :

— Le réseau est un anneau orienté
— Les communications sont FIFO
— Le nombre de sites dans le réseau est inconnu
— Le site ayant la plus petite identité gagne => tous les candidats connaissent l'identité du gagnant.
— 3 états possibles pour un site : En-attente, élu, perdu

Election par constitution de la liste des candidats

```
1  initialisation
2      Etat_i ← non défini
3      ListeCandidats_i ← ∅
4  début de l'élection
5      si Etat_i == non défini alors
6          Etat_i ← candidat
7          ListeCandidats_i ← {i}
8          envoyer ( i ) sur l'anneau
9      fin si
10 réception d'un message
11     recevoir ( j )
12     si Etat_i == candidat alors
13         si j == i alors
14             si i == min ListeCandidats_i alors
15                 Etat_i ← élu
16             sinon
17                 Etat_i ← perdu
18             fin si
19         sinon
20             ListeCandidats_i ← ListeCandidats_i ∪ {j}
21             envoyer ( j ) sur l'anneau
22         fin si
23     sinon
24         Etat_i ← perdu
25         envoyer ( j ) sur l'anneau
26     fin si
```

Principe : Cet algo utilise le fait qu'il est possible de calculer une valeur sur le réseau avec une onde si l'opérateur est binaire associatif et commutatif. Ici, on cherche donc à calculer le minimum des IDs, NON PLUS APRES que la liste des candidats ai été faite, MAIS plutot, au fur et à mesure que l'algo est executé => il n'est plus necessaire de stcoker la liste des candidats. Seul le meilleur candidat (au moment de la reception de la candidature, est conservé).

Election par calcul du minimum

```
1  initialisation
2      Elu_i ← +∞
3  début de l'élection
4      si Elu_i == +∞ alors
5          Elu_i ← i
6          envoyer ( i ) sur l'anneau
7      fin si
8  réception d'un message
9      recevoir ( j )
10     si j == i alors
11         FIN                                   /* S_i est élu */
12     si j < Elu_i alors
13         Elu_i ← j
14         envoyer ( j ) sur l'anneau
15     fin si
```

Hypothèses :
— Communications bi-directionnelles
— Il n'existe pas d'anneau de contrôle permettant d'appliquer les algos précédents

Election dans un réseau quelconque – version centralisée

```
1  initialisation
2      Parent_i ← 0
3      NbMessagesAttendus_i ← nb de voisins
4      Elu_i ← +∞
5  début de la vague
6      Elu_i ← i
7      Parent_i ← i
8      envoyer ( [bleu] ) aux voisins
9  début de l'élection
10     si Parent_i == 0 alors
11         Elu_i ← i
12     fin si
13 réception d'un message bleu
14     recevoir ( [bleu] ) de S_j
15     si Parent_i == 0 alors
16         Parent_i ← j
17         NbMessagesAttendus_i ← NbMessagesAttendus_i − 1
18         si NbMessagesAttendus_i == 0 alors
19             envoyer ( [rouge] Elu_i ) à S_{parent_i}
20         sinon
21             envoyer ( [bleu] ) aux voisins sauf S_j
22         fin si
23     sinon
24         envoyer ( [rouge] Elu_i ) à S_j
25     fin si
26 réception d'un message rouge
27     recevoir ( [rouge] Elu )
28     Elu_i ← min(Elu, Elu_i)
29     NbMessagesAttendus_i ← NbMessagesAttendus_i − 1
30     si NbMessagesAttendus_i == 0 alors
31         si Parent_i == i alors
32             FIN                                  /* l'élu est S_{élu_i}) */
33         sinon
34             envoyer ( [rouge] Elu_i )
35         fin si
36     fin si
```

Principe : Cet algo vise à faire partir une vague depuis chaque site candidat. Lorsqu'un site reçoit plusieurs vagues, il choisit celle du candidat d'identité la plus petite. C'est la technique dite d'extinction. Si un site est atteint par au moins une vague, il ne peut plus devenir candidat et lancer sa propre vague.

Election par extinction (version décentralisée)

```
1  initialisation
2      Parent_i ← 0
3      NbVoisinAttendus_i ← nb de voisins
4      Elu_i ← +∞
5  début de la vague                          /* sur les sites candidats */
6      si Parent_i == 0 alors
7          Elu_i ← i
8          Parent_i ← i
9          envoyer ( [bleu] Elu_i )
10     fin si
11 réception d'un message bleu
12     recevoir ( [bleu] k ) de S_j
13     si Elu_i > k alors
14         Elu_i ← k
           /* on oublie la vague en cours, s'il y en avait une     */
15         Parent_i ← j
16         NbVoisinAttendus_i ← nb de voisins - 1
17         si NbVoisinAttendus_i > 0 alors  /* diffuser une nouvelle vague */
18             envoyer ( [bleu] Elu_i ) aux voisins sauf S_j
19         sinon                                   /* la vague remonte */
20             envoyer ( [rouge] Elu_i ) à S_j
21         fin si
22     sinon
23         si Elu_i == k alors
24             envoyer ( [rouge] Elu_i ) à S_j
25         fin si
26     fin si
27 réception d'un message rouge
28     recevoir ( [rouge] k )
29     si Elu_i == k alors
30         NbVoisinsAttendus_i ← NbVoisinsAttendus_i − 1
31         si NbVoisinsAttendus_i == 0 alors
32             si Elu_i == i alors
33                 FIN                              /* l'élu est S_élu_i */
34             sinon
35                 envoyer ( [rouge] Elu_i ) à S_parent_i
36             fin si
37         fin si
38     fin si
```

5 Détection de terminaisons

Principe : La technique des calculs diffus s'apparente à une vague centralisée. L'arborescence est mouvante : elle est construite et modifiée au gré des activations et désactivations des sites. Ici, ce ne sont pas les voisins qui sont attendus, mais les accusés de réception. Ceux-ci remontent l'arborescence au fur et à mesure que ses extrémités se «fanent». En quelque sorte, les messages rouges sont toujours envoyés par des sites feuilles, avant qu'ils ne quittent l'arborescence. Quand la racine est fanée, l'application est terminée. Ce mécanisme de terminaison est valable pour un unique initiateur.

Algorithme de contrôle implémentant la méthode de détection de terminaison des calculs diffus

```
1  initialisation
2      Parent_i ← 0
3      bilan_i ← 0
4      passif_i ← vrai
5  début d'activité                                   /* sur un seul site */
6      Parent_i ← i
7      passif_i ← faux
8  fin d'activité de l'instance locale de l'app. de base
9      passif_i ← vrai
10 émission d'un message m par l'instance locale de l'app. de base
11     bilan_i ← bilan_i + 1
12     envoyer ( [msg] m ) à S_j
13 réception d'un message lié à l'app. de base
14     recevoir ( [msg] m ) de S_j
15     si Parent_i == 0 alors                        /* message de réveil */
16         passif_i ← faux
17         Parent_i ← j
18     sinon                                  /* message tout de suite acquitté */
19         envoyer ( [acquittement] ) à S_j
20     fin si
21     transmettre le message à l'app. de base locale
22 réception d'un acquittement
23     recevoir ( [acquittement] )
24     bilan_i ← bilan_i − 1
25 (passif_i == vrai) et (bilan_i == 0) et (Parent_i ≠ 0)
       /* il faut quitter l'arborescence                    */
26     si Parent_i == i alors
27         FIN
28     sinon
29         envoyer ( [acquittement] ) à S_{parent_i}
30         Parent_i ← 0
31     fin si
```

Principe : Maintenir, sur chaque site S_j, un vecteur d'entiers relatifs dont la k^eme composante représente le nombre de message emis vers le site k. De même, la j^eme composante représente l'opposé du nombre de messages reçus par Sj. Il est necessaire de gérer les court-circuits du jeton par des messages de «réveil», réactivant les sites après que le jeton les aient trouvés inactifs. Lorsque le jeton arrive sur un site actif, il est arreté. Il est donné au noeud suivant que lorsque le site courant est devenu passif (lorsque le jeton est donné au site suivant, alors le compteur local du site est remit à zéro). Lorsque l'application est terminée, toutes les composantes du jeton-vecteur sont nulles.

Avantages : Cette technique est rapide. Il ne faut qu'un seul tour d'anneau pour permettre au dernier site ayant bloqué le jeton de conclure à la terminaison.

Inconvénients : Cette technique est gourmande en place memoire

Hypothèses :

— Anneau de contrôle (circuit hamiltonien)

Jeton-vecteur

```
1  initialisation
2      pour k = 1 à N faire
3          V_i[k] ← 0
4          VJ_i[k] ← 0
5      fin pour
6      passif_i ← vrai
7      jetonPresent_i ← faux
8      jetonTransmis_i ← faux
9  début d'activité de l'instance locale de l'app. de base
10     passif_i ← faux
11 fin d'activité de l'instance locale de l'app. de base
12     passif_i ← vrai
13 départ du jeton
14     pour k = 1 à N faire
15         VJ_i[k] ← VJ_i[k] + V_i[k]
16         V_i[k] ← 0
17     fin pour
18     jetonTransmis_i ← vrai
19     envoyer ( [jeton] VJ_i ) au successeur sur l'anneau
20 demande d'émission d'un message m vers S_j par l'instance locale de l'app. de base

21     V_i[j] ← V_i[j] + 1
22     envoyer ( m ) à S_j
23 réception d'un message m
24     recevoir ( m )
25     V_i[i] ← V_i[i] − 1
26     transmettre le message à l'app. de base
27 réception du jeton
28     recevoir ( [jeton] VJ )
29     VJ_i ← VJ
30     jetonPresent_i ← vrai
31 (jetonPresent_i == vrai) et (passif_i == vrai)
32     pour k = 1 à N faire
33         VJ_i[k] ← VJ_i[k] + V_i[k]
34         V_i[k] ← 0
35     fin pour
36     si (jetonTransmis_i == vrai) et (VJ_i[k] == 0 ∀k, 1 ≤ k ≤ N) alors
37         FIN pour S_i
38     fin si
39     envoyer ( [jeton] VJ_i ) au successeur sur l'anneau
40     jetonPresent_i ← faux
41     jetonTransmis_i ← vrai
```

6 Detection des interblocages

Détection de l'interblocage dans le contexte général – Application de base

```
1  initialisation
2      NumRequête_i ← 0
3  émission d'une requête
4      NbRépAttendues_i ← nb de destinataires
5      P_i ← prédicat correspondant à la réponse attendue
6      RépR_i ← ∅
7      ReqR_i ← ∅
8      Etat_i ← en_attente
9      NumRequête_i++
10     envoyer ( i, requête, p ) aux destinaires
11 réception d'une requête
12     recevoir (j, requête, p)
13     si Etat_i ≠ en_attente alors
14         envoyer ( réponse, q ) à S_j
15     sinon
16         ReqR_i ← ReqR_i ∪ ⟨j, requête, q⟩
17     fin si
18 réception d'une réponse
19     recevoir ( réponse, q )
20     si q == NumRequête_i et Etat_i == en_attente alors
21         NbRépAttendues_i- -
22         RépR_i ← RépR_i ∪ {réponse}
23         si P_i(RépR_i) ou NbRépAttendues_i == 0 alors
24             Etat_i ← actif
25             pour tout ⟨j, requête, q⟩ ∈ ReqR_i faire
26                 envoyer ( réponse, q ) à S_j
27             fin pour
28         fin si
29     fin si
```

7 Exclusion mutuelle

Exclusion mutuelle avec permission individuelle, site S_i

```
   /* Attendus_i = {S_1, ..., S_N}                                   */
 1 initialisation
 2     RéponsesDifférées_i ← ∅                  /* requêtes en souffrance */
 3     Reçus_i ← ∅                               /* autorisations reçues */
 4     h_i ← 0
 5     hdem_i ← 0                          /* date de la dernière demande */
 6     état_i ← repos
 7 réception demande de section critique
 8     recevoir ( [demandeSC] ) de l'app. de base
 9     h_i ← h_i + 1
10     hdem_i ← h_i
11     état_i ← demandeur
12     envoyer ( [requête] h_i ) à tous les sites sauf S_i
13     Reçus_i ← ∅
14 réception fin de section critique
15     recevoir ( [finSC] ) de l'app. de base
16     h_i ← h_i + 1
17     pour tout S_k ∈ RéponsesDifférées_i faire
18         envoyer ( [permission] h_i ) à S_k
19     fin pour
20     état_i ← repos
21     RéponsesDifférées_i ← ∅
22 réception d'une requête
23     recevoir ( [requête] h ) de S_j
24     h_i ← max(h_i, h) + 1
25     si (état_i ≠ repos) et ((hdem_i, i) <_2 (h, j)) alors
26         RéponsesDifférées_i ← RéponsesDifférées_i ∪ {S_j}
27     sinon
28         envoyer ( [permission] h_i ) à S_j
29     fin si
30 réception d'une permission
31     recevoir ( [permission] h ) de S_j
32     h_i ← max(h_i, h) + 1
33     Reçus_i ← Reçus_i ∪ {S_j}
34     si (Reçus_i == Attendus_i) alors
35         envoyer ( [débutSC] ) à l'app. de base
36         état_i ← satisfait
37     fin si
```

Exclusion mutuelle avec répartition d'une file d'attente, site S_i

```
1  initialisation
2      Tab_i[k] ← (libération, 0) pour tout k ∈ {1, ..., N}
3      h_i ← 0
4  réception demande de section critique
5      recevoir ( [demandeSC] ) de l'app. de base
6      h_i ← h_i + 1
7      Tab_i[i] ← (requête, h_i)
8      envoyer ( [requête] h_i ) à tous les autres sites
9  réception fin de section critique
10     recevoir ( [finSC] ) de l'app. de base
11     h_i ← h_i + 1
12     Tab_i[i] ← (libération, h_i)
13     envoyer ( [libération] h_i ) à tous les autres sites
14 réception d'une requête
15     recevoir ( [requête] h ) de S_j
16     h_i ← max(h_i, h) + 1
17     Tab_i[j] ← (requête, h)
18     envoyer ( [accusé] h_i ) à S_j
19     si (Tab_i[i].type == requête) et ((Tab_i[i].date, i) <_2 (Tab_i[k].date, k) pour
         tout k ≠ i) alors
20         envoyer ( [débutSC] ) à l'app. de base
21     fin si
22 réception d'une libération
23     recevoir ( [libération] h ) de S_j
24     h_i ← max(h_i, h) + 1
25     Tab_i[j] ← (libération, h)
26     si (Tab_i[i].type == requête) et ((Tab_i[i].date, i) <_2 (Tab_i[k].date, k) pour
         tout k ≠ i) alors
27         envoyer ( [débutSC] ) à l'app. de base
28     fin si
29 réception d'un accusé
30     recevoir ( [accusé] h ) de S_j
31     h_i ← max(h_i, h) + 1
32     si Tab_i[j].type ≠ requête alors
33         Tab_i[j] ← (accusé, h)
34     fin si
35     si (Tab_i[i].type == requête) et ((Tab_i[i].date, i) <_2 (Tab_i[k].date, k) pour
         tout k ≠ i) alors
36         envoyer ( [débutSC] ) à l'app. de base
37     fin si
```