# CHAPITRE VIII

## AUTO-STABILISATION

*version du 1er décembre 2003*

## 1 Problématique

Afin de cerner le concept d'algorithme auto-stabilisant, nous allons étudier le comportement d'un composant actif de réseaux, le commutateur.

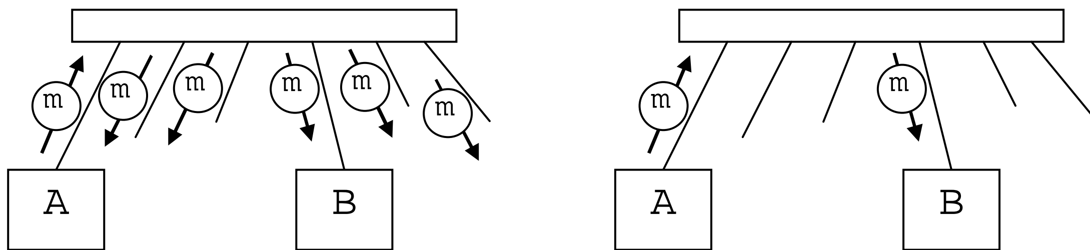



Figure 8.1 Fonctionnement d'un concentrateur et d'un commutateur

La figure 8.1 présente deux composants actifs de réseau : le concentrateur et le commutateur. Tous les deux sont dotés de ports (éventuellement) connectés soit à des machines soit à d'autres composants de réseaux. Lorsque le concentrateur (représenté à gauche) reçoit une trame `m` à destination de la machine `B` - la machine est ici désignée par son adresse MAC -, il retransmet la trame sur tous ses autres ports. Cette façon de procéder est simple mais inefficace car la bande passante des ports est consommée inutilement et de plus dans le cas du protocole Ethernet, le risque de collision est augmenté.

De son côté, le commutateur (représenté à droite) ne retransmet la trame que sur le port à partir duquel il "pense" pouvoir atteindre la machine `B`. Mais de quelle façon connait-il ce port ? La réponse réside dans le mécanisme d'apprentissage dont est pourvu le commutateur. Le commutateur est doté d'une table associative (notée `Table` dans la suite) dans laquelle chaque cellule comporte deux champs : une adresse de machine et un numéro de port. Cette table est initialement vide. Nous décrivons ci-dessous le principe d'acheminement d'une trame.

Lorsqu'une trame arrive sur un port, le commutateur met à jour la table en ajoutant l'association entre l'adresse de l'émetteur et le port de réception (et en supprimant éventuellement une association erronée). Puis il recherche une association pour le destinataire et envoie la trame sur le port trouvé en cas de succès. En cas d'échec, il diffuse la trame sur tous les ports (excepté le port de réception). Nous avons indiqué ci-dessous une version algorithmique de ce mécanisme.

```
Sur_réception_de(port,(@emet,@dest,m))
Début
     portbis = recherche(Table,@emet);
     Si portbis != port  Alors
          Si portbis != NULL Alors
               Supprime(Table,@emet);
          Finsi
          Ajoute(Table,(@emet,port));
     Finsi
     portbis = recherche(Table,@dest);
     Si portbis != NULL  Alors
          envoyer_à(portbis,(@emet,@dest,m));
     Sinon
          Pour tout portbis!=port faire
               envoyer_à(portbis,(@emet,@dest,m));
          Finpour
     Finsi
Fin
```

Ainsi très rapidement, le commutateur localise toutes les machines sur le réseau et les trames ne sont plus diffusées. Qu'arrive-t-il si la table ne reflète plus la topologie du réseau ? Ceci peut survenir à la suite d'une perturbation temporaire du commutateur ou encore plus simplement en raison d'une modification des connexions par un administrateur réseau.

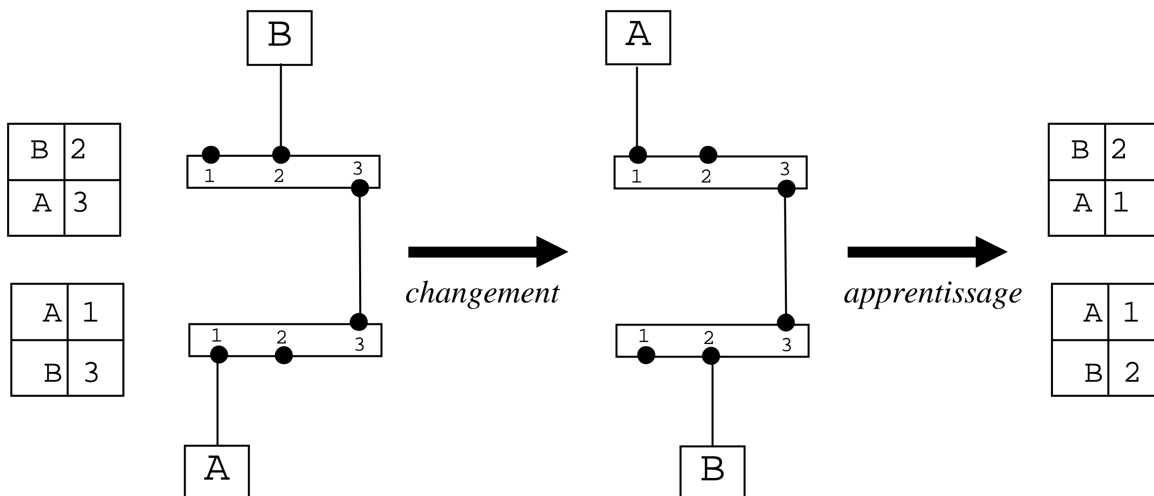



Figure 8.2 Une modification de la topologie et ses conséquences

Sur la figure 8.2, le réseau est constitué de deux commutateurs interconnectés. Les deux machines `A` et `B` ont été déplacées et reconnectées. Avec le mécanisme d'apprentissage, chaque commutateur corrige rapidement le port de sortie associé à la machine qu'on lui a nouvellement connectée. Cependant, le commutateur supérieur pense que `B` est connectée sur son port `2` et le commutateur inférieur pense que `A` est connectée sur son port `1`. Aussi si `B` et `A` veulent communiquer, chacun des commutateurs envoie les trames vers une voie "sans issue". Le mécanisme d'apprentissage est alors impuissant car aucune trame ne circule sur les ports n° `3`.

Nous sommes maintenant en mesure d'énoncer la propriété fondamentale d'un algorithme auto-stabilisant. Appelons "configuration", un état global de l'algorithme et de son environnement.

Sans changement ultérieur de l'environnement et partant d'une configuration quelconque, un algorithme auto-stabilisant atteint en un nombre fini d'actions une configuration cohérente, c'est à dire une configuration où les fonctionnalités de l'algorithme seront garanties dans le futur de son exécution.

Remarquons que certaines informations sont permanentes et ne doivent pas être incluses dans la configuration. Dans notre exemple, la numérotation des ports et les adresses des machines sont des informations permanentes. De même, nous supposons que le code de l'algorithme n'est jamais corrompu (par contre le compteur ordinal peut pointer n'importe où dans le code).

Il est possible de transformer le mécanisme de notre commutateur pour éviter la situation décrite plus haute. Pour cela attachons une durée de validité à chaque entrée de la table; cette durée est réinitialisée à chaque fois qu'on reçoit une trame de la machine sur le port associé. Lorsque l'entrée n'est plus valide, elle est supprimée de la table. Avec ce nouveau mécanisme, la gestion du commutateur est-elle auto-stabilisante ? Deux réponses sont possibles, selon ce qu'on considère être la fonctionnalité du commutateur :

- **Acheminer des messages** Alors le commutateur est auto-stabilisant, puisqu'une fois les adresses erronées éliminées, il ne peut y avoir que des adresses correctes dans sa table. Donc soit un message est correctement routé, soit il est diffusé. Dans les deux cas, les messages arrivent à destination.
- **Maintenir une table de routage** Alors le commutateur n'est pas auto-stabilisant car une machine qui ne transmet pas de message pendant un temps supérieur au délai de validité n'est plus référencée dans la table.

On distingue usuellement deux types de tâche pour les algorithmes auto-stabilisants :

- Les tâches statiques : dans une configuration correcte, l'algorithme ne modifie plus les données à maintenir. Le maintien d'une table routage est une tâche statique.
- Les tâches dynamiques : à partir d'une configuration correcte, un service est assuré de manière attendue. La gestion de l'exclusion mutuelle entre sites est une tâche dynamique.

Dans la deuxième section, nous présentons la construction d'un arbre de plus courts chemins qui peut être utilisé par un algorithme de routage. Puis nous montrons comment émuler une mémoire virtuelle répartie avec variables propriétaires. Enfin, en nous appuyant sur l'algorithme précédent, on montrera comment assurer l'exclusion mutuelle entre des sites organisés en anneau. Nous recommandons au lecteur intéressé l'excellent ouvrage de Shlomi Dolev [Dol00].

# 2 Routage auto-stabilisant [Dol89]

## *2.1 Principe et réalisation de l'algorithme*

Nous désirons construire une table de routage au dessus du graphe de communication. On remarque d'abord que l'on peut se restreindre à une destination particulière, la généralisation étant immédiate. Nous appellerons $i_0$ le site à atteindre. L'algorithme a pour objectif de construire un arbre de plus courts chemins vers la destination.

Pour ce faire, chaque site `i` maintient deux variables significatives :

- un tableau $dist_i$ indicé par `i` et ses voisins qui indique la distance supposée à $i_0$
- $père_i$ le voisin à emprunter pour y parvenir.

L'ensemble des voisins est représenté par $voisins_i$ (une information permanente). Le principe de l'algorithme est extrêmement simple. Chaque site exécute une boucle infinie et à chaque tour de boucle :

1. Il calcule en fonction des distances supposées de ses voisins, sa distance minimale à $i_0$ et simultanément son meilleur père.
2. Il envoie à ses voisins sa distance.
3. Il arme un time-out (information permanente) avant d'entamer le tour suivant.

A la réception d'une distance, le site met à jour la cellule appropriée de son tableau.

<u>Variables du site `i`</u>

- $voisins_i$ : constante contenant l'ensemble des voisins `i` dans le graphe de communication.
- $time_out_i$ : constante contenant la valeur d'un délai de suspension.
- $B_i$ : constante contenant un majorant strict du diamètre du réseau.
- $dist_i[voisins_i \cup \{i\}]$ : tableau des distances à $i_0$.
- $père_i$ : identité du prochain noeud sur la route.
- $temp_i$ : variable temporaire.

Algorithme du site `i`

Le service est ici un processus utilitaire qui tourne en tâche de fond.

```
router()
Début
     Tant que VRAI faire
          Si (i==i0) Alors
               disti[i]=0;
               pèrei=i0; // par convention
          Sinon
               disti[i]=Bi;
               Pour tout tempi ∈ voisinsi faire
                    Si disti[tempi]<disti[i]-1 Alors
                         perei=tempi;
                         disti[i]=disti[tempi]+1;
                    Finsi
               Finpour
          Finsi
          Pour tout tempi ∈ voisinsi faire
               envoyer_à(tempi,disti[i]);
          Finpour
          Armer(time_outi)
          Attendre();
     Fin tant que
Fin

sur_réception_de(j,d)
Début
     disti[j]=d;
Fin
```

La figure 8.3 décrit une exécution possible de cet algorithme (le site `1` est la destination). Nous faisons les hypothèses suivantes sur la configuration initiale :

- Le compteur ordinal du site `1` est en début de boucle.
- Le compteur ordinal des autres sites est positionné avant l'envoi des messages.
- Les valeurs initiales du père et de la distance supposée sont telles qu'indiquées sur le premier schéma.
- Le réseau ne contient pas de messages.

De plus, nous supposons que dans la suite de l'exécution, un message envoyé par un site arrive à destination avant l'expiration du time-out sur le site destinaire (ce qui revient à une exécution synchrone, voir le chapitre 3). Les voisins sont examinés par ordre croissant dans la boucle du choix du père.

On remarque les sites se stabilisent de proche en proche en partant de la destination. Nous allons formaliser cet argument pour établir la correction de l'algorithme au prochain paragraphe.

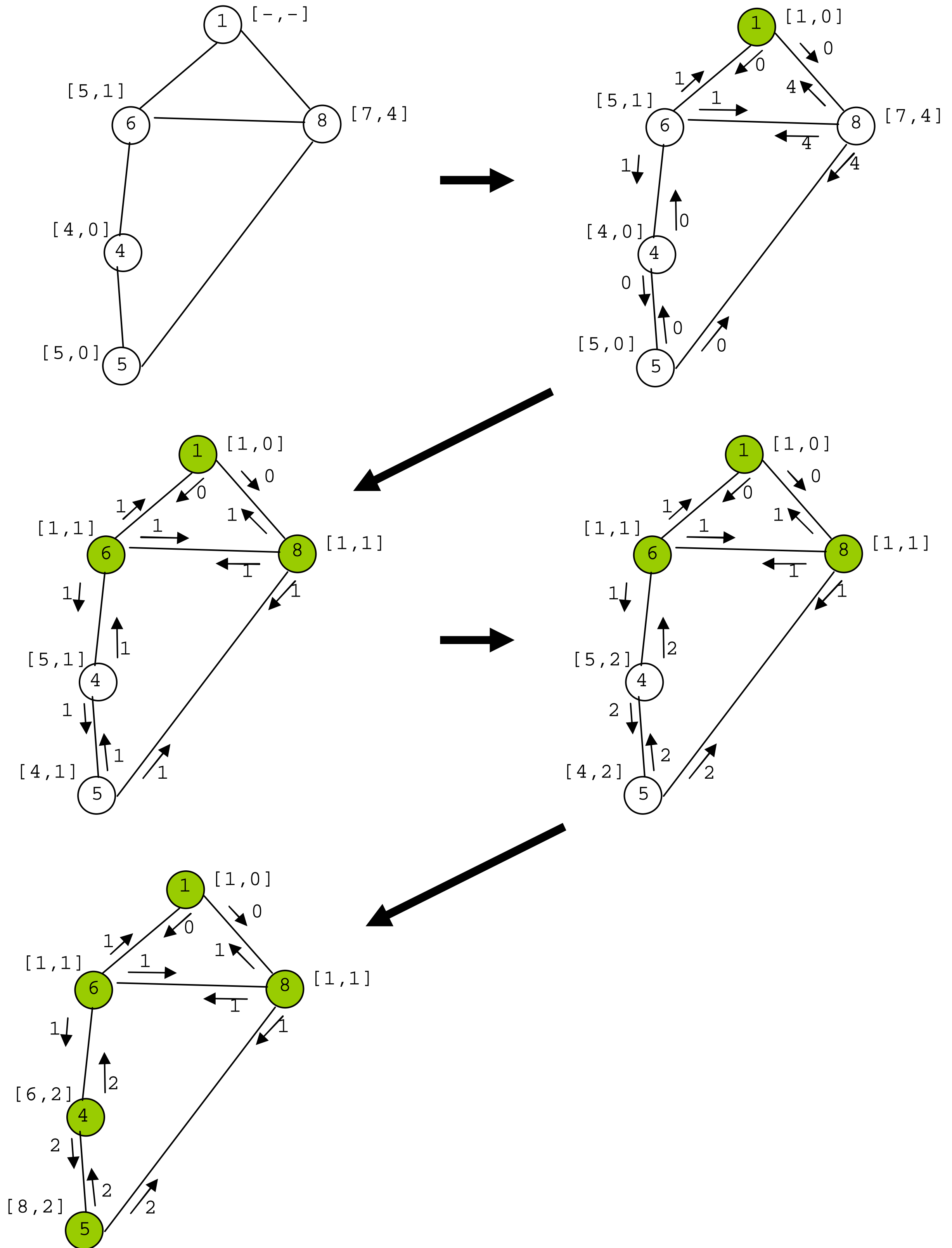



Figure 8.3 Une exécution du routage auto-stabilisant

### 2.2 Preuve de l'algorithme

Nous introduisons d'abord quelques concepts associés à une exécution.

Une exécution est divisée en rondes. Chaque ronde débute à la fin de la ronde précédente et se termine lorsque :

- tous les processus ont exécuté au moins un tour complet de leur boucle ce qu'on appelle la première demi-ronde
- et les messages associés à ce tour de boucle ont été reçus.

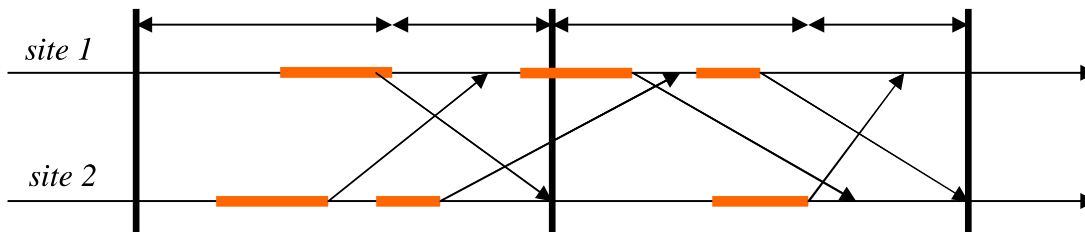



Figure 8.4 Deux rondes du routage auto-stabilisant

Sur la figure 8.4, nous avons représenté deux rondes avec leur découpage en demi-rondes. Les traits oranges correspondent à un tour de boucle du processus `router()`. Dans l'exécution présentée, les time-out sont comptés irrégulièrement (car la correction de l'algorithme ne repose pas sur une valeur particulière de temporisation). Notons d'abord qu'au cours d'une ronde, un site peut exécuter plusieurs tours de boucle et d'autre part que nous ne comptons pas dans une ronde, un tour de boucle débuté au cours de la ronde précédente.

Etant donnée une configuration, on appelle "une valeur fausse", une distance dans une variable ou dans un message qui ne correspond pas à la distance réelle. La `ppvf` correspond à la plus petite valeur fausse de la configuration. Par convention, s'il n'existe pas de valeur fausse on pose $\mathtt{ppvf} = +\infty$.

Nous établissons maintenant la propriété clef de cet algorithme.

<u>Propriété</u>

1. Après $k+1$ rondes de l'algorithme, on a $\mathtt{ppvf} > k$
2. Après $k$ rondes et une demi-ronde, l'ensemble des noeuds à une distance inférieure ou égale à $k$ est organisée en un arbre de plus courts chemins (défini par $\mathtt{père}_i$ et $\mathtt{dist}_i[i]$).

<u>Preuve</u> (par récurrence)

Notons $\mathtt{ppvf}^{(k)}$, la `ppvf` à la fin de la $k^{ème}$ ronde et $\mathtt{ppvf}^{(0)}$ la `ppvf` initiale.

Evidemment $\mathtt{ppvf}^{(0)} \geq 0$.

Montrons que $\mathtt{ppvf}^{(k+1)} \geq \mathtt{ppvf}^{(k)}+1$

En effet, au cours de la $k+1^{ème}$ ronde, la distance d'un noeud lorsqu'elle est recalculée est soit exacte, soit supérieure ou égale à $\mathtt{ppvf}^{(k)}+1$. D'autre part à la fin de cette ronde ne circulent plus que des distances recalculées au cours de cette ronde puisque chaque canal a reçu une valeur recalculée (par définition d'une ronde). Ceci achève la preuve du premier point.

Après la première demi-ronde, la destination a ses variables, distance et père, correctement positionnées.

Supposons la propriété 2. établie à la k+1$^{ème}$ ronde et examinons la k+2$^{ème}$ ronde. Au cours de cette ronde, la `ppvf` est strictement supérieure à `k`. Donc les sites à distance inférieure ou égale à `k` conservent leurs variables, père et distance, inchangées. D'autre part, les messages qui définissent la deuxième demi-ronde de la ronde `k+1` ont été reçus par les sites à distance `k+1`. A la fin de la première demi-ronde de la ronde `k+2`, ces sites évalueront leur distance et choisiront leur père de manière correcte en raison du minorant du point 1. sur la `ppvf`.

Autrement dit, si `D` est le diamètre du graphe, au bout d'au plus `D+1/2` rondes, l'arbre est établi.

# 3 Gestion de mémoire virtuelle répartie [Dol97]

## *3.1 Principe de l'algorithme*

Dans l'algorithme précédent, les hypothétiques messages qui circuleraient sur le réseau n'empêche pas celui-ci de se stabiliser. Cependant la conception de la plupart des algorithmes auto-stabilisants se fait en supposant que les processus partagent une mémoire virtuelle répartie et plus précisément une gestion de variables propriétaires (voir le chapitre 6). Comme nous le verrons dans l'algorithme de la prochaine section, cela facilite grandement la conception des algorithmes.

Mais cette supposition n'a d'intérêt que si la gestion de la mémoire est elle-même auto-stabilisante et c'est donc l'objectif de la présente section. Avant toute chose, il faut savoir que dans un environnement réseau purement asynchrone, il n'existe pas de solution auto-stabilisante à ce problème [Gou91]. Heureusement moyennant des hypothèses faibles, ce résultat d'impossibilité n'est plus vérifié. Dans notre cas, nous supposons que chaque canal bidirectionnel contient un nombre de messages strictement inférieur à une borne `B`.

La solution s'obtient par modifications successives du protocole de lecture à la demande. Dans le protocole initial, le lecteur envoie une requête au propriétaire et attend la valeur en retour. En raison du caractère quelconque de la configuration initiale, il se peut que le lecteur attende une réponse que le propriétaire n'enverra jamais. Aussi la première modification consiste à faire circuler un jeton qui "porte" à la fois la requête et la réponse. Le lecteur attend un premier passage du jeton qui correspond à l'émission de sa requête et un deuxième passage pour recevoir la réponse.

Ceci ne résout pas encore le problème puisque le jeton peut être absent de la configuration initiale. Aussi à l'aide d'un time-out, si le propriétaire ne voit passer pas de jeton, il le réémet avec bien sûr dans ce cas la possibilité de plusieurs "jetons" circulant simultanément. Ce remède introduit un nouveau problème. Comment le lecteur va-t-il reconnaitre que le message qui arrive est le jeton ?

Pour cela, les messages sont numérotés par un compteur qui s'incrémente modulo `B` (le choix de ce modulo s'expliquera lors de la preuve). Chacun des sites a son propre compteur. Lorsque le lecteur voit arriver un message avec une nouvelle valeur, il reconnait le jeton. De même lorsque le propriétaire voit arriver un message avec sa propre valeur, il reconnait le jeton et incrémente son compteur avant de renvoyer la valeur. Le propriétaire supprime les messages différents du jeton.

Nous décrivons maintenant l'algorithme, puis nous illustrons son fonctionnement nominal avant d'établir la preuve de la stabilisation. L'algorithme se généralise facilement au cas de plusieurs lecteurs et de plusieurs variables. De plus tel qu'il est écrit, il supporte des canaux à pertes du moment qu'un message réémis indéfiniment finit par être reçu.

### 3.2 Description et exemple d'exécution

<u>Variables du site distant</u> i

- $état_i$ : état du service. Cette variable prend une valeur parmi (repos, prem_att, sec_att).
- $h_i$ : compteur du lecteur
- $val_i$ : valeur à renvoyer à la lecture

<u>Algorithme du site distant</u> i
La lecture consiste à se mettre en (première) attente et attendre de repasser au repos pour renvoyer la valeur.

```
read(x)
Début
     état_i=prem_att;
     Attendre(état_i==repos);
     renvoyer(val_i);
Fin
```

A la réception d'une message, on teste d'abord s'il s'agit du jeton. Dans ce cas, on met à jour son compteur, puis si une lecture est en première attente, elle passe en seconde attente et si elle est en seconde attente, elle passe au repos en récupérant la valeur courante de l'objet. Dans tous les cas le message est renvoyé.

```
sur_réception_de(j,(v,h))
Début
     Si (h_i!=h) Alors
          h_i=h;
          Si (état_i == prem_att) Alors
               état_i=sec_att;
          Sinon si (état_i == sec_att) Alors
               val_i=v;
               état_i=repos;
          Finsi
     Finsi
     envoyer_à(j,(acq,h));
Fin
```

<u>Variables du site propriétaire</u> j

- $time_out_j$ : constante contenant la valeur d'un délai de suspension.
- $h_j$ : compteur du propriétaire
- $x_j$ : valeur de l'objet x

```
Algorithme du site propriétaire j
```

L'écriture est purement locale.

```
write(x,v)
Début
     xj=v;
Fin
```

Ce processus utilitaire a pour but d'envoyer la valeur au site distant si un time-out expire.

```
régénérer()
Début
     Tant que VRAI faire
          envoyer_à(i,(xj,hj));
          Armer(time_outj);
          Attendre();
     Fin tant que
Fin
```

A la réception d'un message qui porte le compteur (i.e. reconnu comme un jeton), on incrémente celui-ci et on le renvoie. Puis on envoie la valeur et le compteur et on réarme le time-out pour éviter des envois superflus.

```
sur_réception_de(i,(acq,h))
Début
     Si (hj==h) Alors
               hj = hj+1 %B;
               envoyer_à(i,(xj,hj));
               Armer(time_outj);
     Finsi
Fin
```

Nous appelons dans la suite, une configuration nominale initiale, une configuration telle que les deux compteurs sont égaux, qu'il y a au moins un message dans le canal bidirectionnel et que tous les messages ont une valeur de compteur identique à celle des sites. Le message le plus "proche" du site propriétaire est appelé le jeton. La configuration (a) de la figure 8.5 décrit une telle situation. Avant de continuer, le lecteur est invité à vérifier que sur toutes les configurations de la figure 8.5, une réémission de message (suite à une expiration du time-out) ne change pas la situation décrite.

(b) Le jeton est arrivé sur le propriétaire qui incrémente son compteur ($\oplus$ désigne l'addition modulo B) et renvoie le jeton. Les messages présents sur le réseau s'ils arrivent sur le lecteur ne sont pas reconnus comme étant le jeton.

(c) Le jeton se dirige vers le lecteur suivi d'éventuelles copies (dues à l'expiration du time-out). Les autres messages sont absorbés par le propriétaire.

(d) Le jeton est le message le plus proche du lecteur.

(e) Le jeton est arrivé sur le lecteur qui met à jour son compteur.

(f) Les copies récentes du jeton ne sont pas reconnues par le lecteur; les vieilles copies sont absorbées et nous retrouvons la situation (a) avec les compteurs "incrémentés" de `1`.

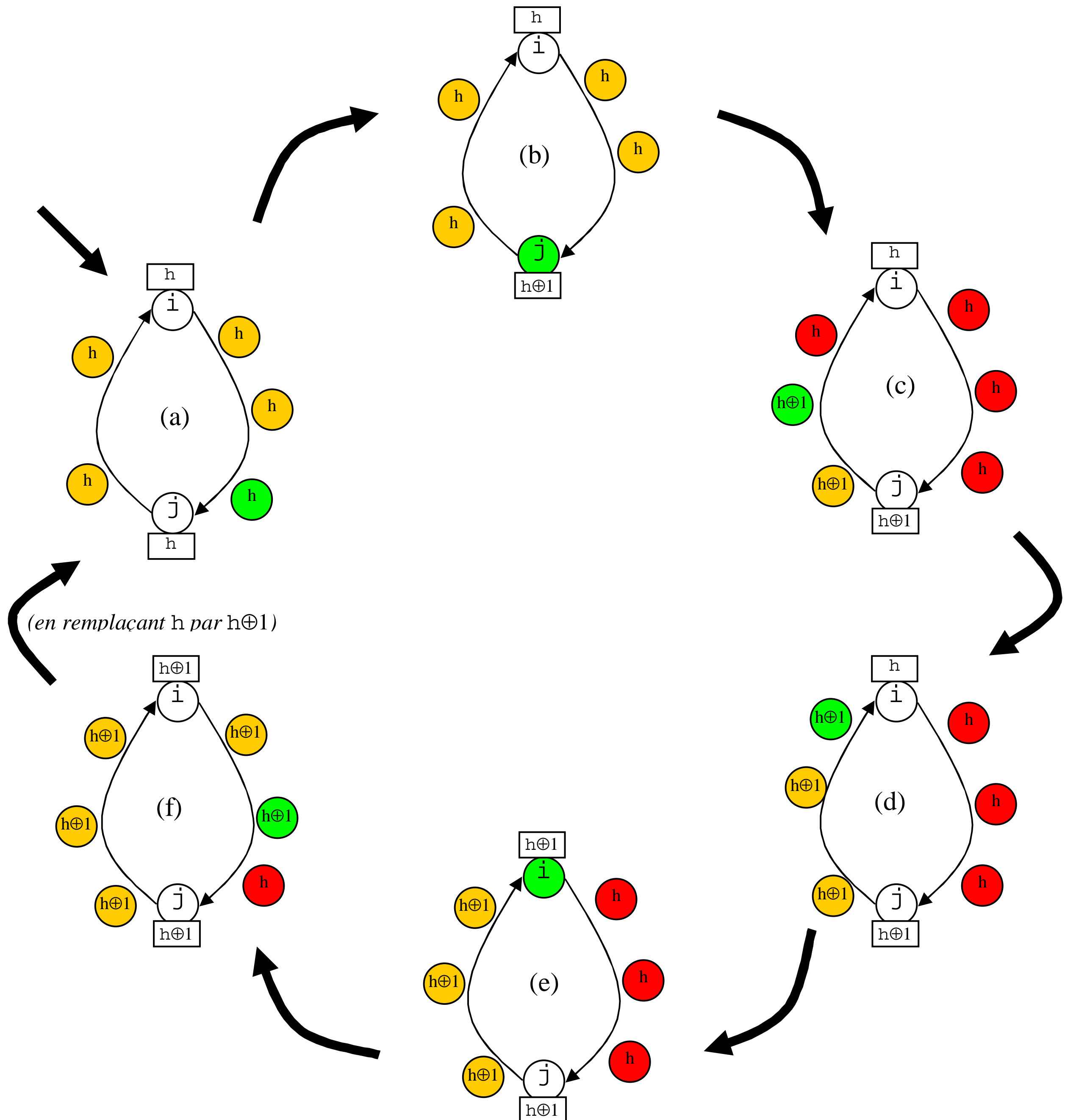



Figure 8.5 Circulation du jeton (une fois la phase de stabilisation passée)

### *3.3 Correction de l'algorithme*

Comme l'avons vu sur l'exemple précédent, une fois atteinte une configuration nominale initiale, l'algorithme est semblable au protocole de lecture à demande. Il ne nous reste qu'à démontrer que partant d'une configuration quelconque, on atteint une configuration nominale initiale.

Nous procédons par étapes.

1. Le compteur du propriétaire ne reste jamais bloqué. Supposons qu'il n'en soit pas ainsi. Un message estampillé avec cette valeur sera émis à l'expiration du time-out puisque le propriétaire ne reconnait jamais le jeton. Ce message passe le lecteur et revient au propriétaire qui le reconnait comme jeton et incrémente son compteur. D'où la contradiction.
2. Il y a moins de `B` messages dans la configuration initiale. Donc une valeur de compteur est absente. D'après le premier point, on atteint une configuration où le compteur du propriétaire a cette valeur (ce peut être la configuration initiale). Dans cette configuration, un message est envoyé avec la valeur de ce compteur et il est le seul à avoir cette propriété.
3. Tous les messages qui le précèdent sont supprimés par le propriétaire et ceux qui le suivent portent la même valeur de compteur. Lorsqu'il arrive sur le lecteur, celui-ci prend cette valeur. Enfin lorsqu'il est le message le plus proche du propriétaire, nous retrouvons la configuration (a) de la figure 8.5.

# 4 Exclusion mutuelle [Dij73]

## *4.1 Présentation*

Le lecteur attentif aura remarqué que le protocole précédent assure implicitement une exclusion mutuelle entre le propriétaire et le lecteur. On désire généraliser ce principe pour obtenir l'exclusion mutuelle entre un ensemble de sites. C'est l'objectif de l'algorithme de Dijkstra qui a de plus la particularité d'être l'agorithme d'où a émergé le concept d'auto-stabilisation.

Tout d'abord nous organisons les N sites en un anneau "unidirectionnel" avec un site distingué $i_0$ (l'équivalent du propriétaire). Chaque site est pourvu d'un compteur. Nous faisons l'hypothèse simplificatrice que chaque site peut lire la valeur du compteur du site précédent sur l'anneau. Dans les deux derniers paragraphes, nous lèverons cette hypothèse.

La possibilité d'entrer en section critique est matérialisée de la façon suivante :

- Le site distingué peut entrer en section critique, lorsque son compteur est égal au compteur précédent.
- Un autre site peut entrer en section critique lorsque son compteur est différent du compteur précédent.

La sortie de section critique se définit ainsi :

- Le site distingué incrémente son compteur modulo N+1 (la valeur du modulo se comprendra lors de la preuve).
- Un autre site recopie la valeur du compteur précédent dans son compteur.

Evidemment, si l'application n'est pas intéressée par la section critique, le service effectue la sortie de section critique immédiatement. Avant de spécifier l'algorithme, nous donnons un exemple de fonctionnement nominal et une phase de stabilisation.

De manière analogue à l'algorithme précédent, une configuration nominale initiale est une configuration dans laquelle tous les compteurs sont égaux.

Ainsi les schémas (a) et (f) de la figure 8.6 décrivent des configurations nominales initiales. Les schémas intermédiaires montrent que chaque site sur l'anneau peut successivement entrer en section critique. De plus, aucun autre site ne peut entrer simultanément en section critique. Les propriétés de sûreté et de vicacité de l'exclusion mutuelle sont ainsi vérifiées si on démarre d'une configuration initiale nominale. Sur cette figure et la suivante, les sites qui ont la possibilité d'entrer en section critique sont colorés en vert et le site distingué est cerclé d'un trait gras.

Il nous restera à montrer que de toute configuration on atteint une configuration initiale nominale. Pour comprendre ce qui garantit cette "convergence", nous déroulons un exemple de la phase de stabilisation sur la figure 8.7.

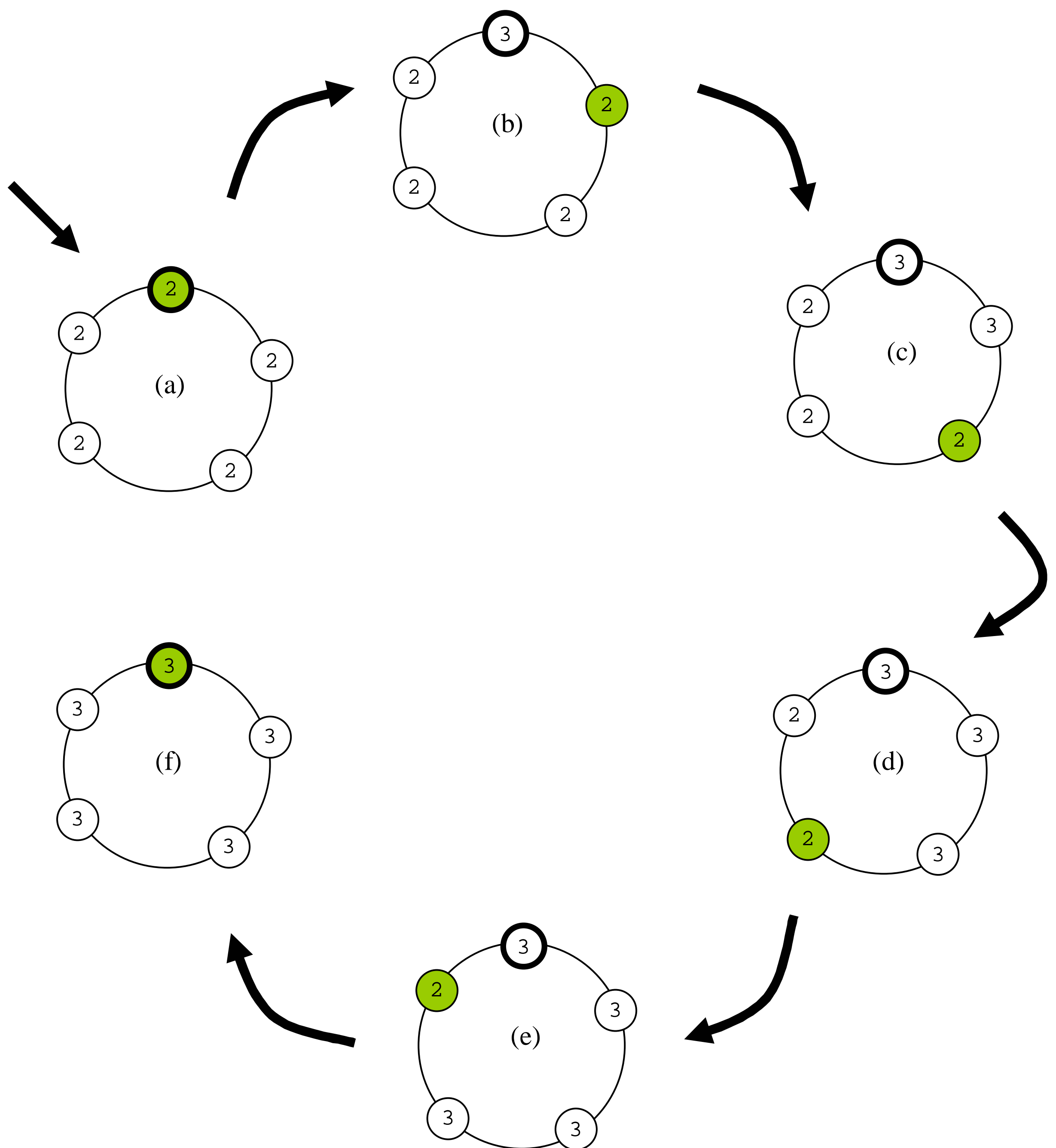



Figure 8.6 Fonctionnement nominal

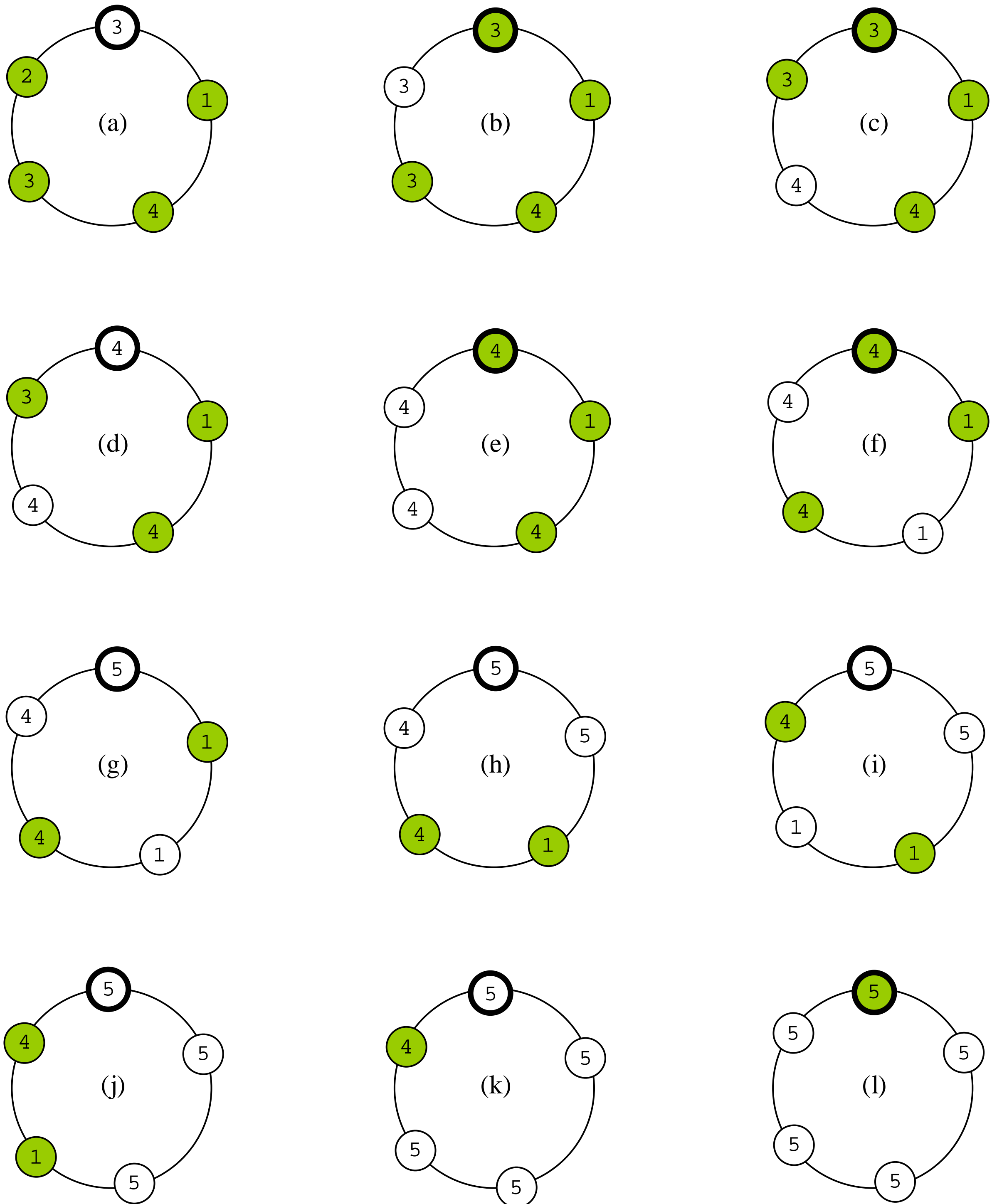



Figure 8.7 Phase de stabilisation

Au cours de la stabilisation, le site distingué incrémente son compteur en (d) et en (g). La valeur prise en (g) est absente sur les autres sites. Aussi la prochaine fois, que le site distingué incrémente son compteur (après la configuration (l)), les autres sites se seront transmis cette valeur et on aura atteint une configuration nominale initiale. La preuve formelle suit la spécification de l'algorithme.

### 4.2 Spécification de l'algorithme

Lorsque l'application d'un site veut exécuter une fonction en section critique, elle appelle la primitive `SC(f)` où `f` est la fonction à exécuter. Le service est doté d'un processus `circuler()` qui tourne en arrière plan.

Variables du site `i`

- $c_i$ : compteur du site i.
- $cprec_i$ : compteur du site précédent i sur l'anneau. Il n'est accédé qu'en lecture par le site `i`.
- $f_i$ : fonction à exécuter en section critique
- $time_out_i$ : constante contenant la valeur d'un délai de suspension.

Algorithme du site `i`

Le processus qui fait circuler le droit d'entrer en section critique, teste d'abord la possibilité d'entrée en section critique. Si tel est le cas, il appelle la fonction prévue et remet la fonction à `NULL` après exécution. Il fait ensuite circuler le droit. Nous faisons ici l'hypothèse que le mécanisme d'appel de fonction applicative ne peut aborter le processus de service et que ce mécanisme teste si l'adresse est nulle.

```
circuler()
Début
     Tant que VRAI faire
          Armer(time_outi);
          Attendre();
          Si   (i==i0)&&(ci==cpreci)||
               (i!=i0)&&(ci!=cpreci) Alors
               fi();
               fi=NULL;
               Si (i==i0) Alors
                    ci = ci+1 %(N+1);
               Sinon
                    ci = cpreci;
               Finsi
          Finsi
     Fin tant que
Fin
```

Pour exécuter une fonction `f` en section critique un processus positionne la variable de fonction de manière appropriée et attend l'indication d'exécution.

```
SC(f)
Début
      fi=f;
      Attendre(fi==NULL);
Fin
```

### 4.3 Correction de l'algorithme

Nous remarquons d'abord que la variable $f_i$ sera nécessairement soit positionnée à `NULL` soit à la fonction à exécuter après l'obtention du premier droit. Seul le premier appel à $f_i()$ peut donc être incorrect. Il suffit alors de démontrer que l'on atteint nécessairement une configuration initiale nominale et se placer après l'obtention du premier droit sur chaque site.

Nous procédons en plusieurs étapes.

1. Le compteur du site distingué ne reste jamais bloqué. Supposons qu'il n'en soit pas ainsi. Alors le site suivant ne peut plus faire qu'une mise à jour de sa variable. Le suivant de ce site ne peut faire que deux mises à jour, etc. On obtient donc une configuration où aucun site ne peut faire de mise à jour. En raisonnant successivement sur les sites non distingués, on conclut que toutes les valeurs sont identiques. Mais dans ce cas, le site distingué peut faire une mise à jour, d'où la contradiction.
2. Il y a moins de `N+1` valeurs différentes dans la configuration initiale. Donc une valeur de compteur `val` est absente. D'après le premier point, on atteint une configuration $conf_1$ où le compteur du site distingué a cette valeur (ce peut être la configuration initiale).
3. Examinons maintenant la configuration $conf_2$ qui suit $conf_1$ dans laquelle le site distingué peut à nouveau modifier son compteur. Puisque `val` est absente des compteurs des sites non distingués, un raisonnement par récurrence (en "remontant" l'anneau à partir du site distingué) montre que chacun d'entre eux doit positionner au moins une fois son compteur à `val`. Un autre raisonnement par récurrence (cette fois-ci dans le sens de l'anneau) montre que ces sites ne peuvent ensuite le modifier tant que le site distingué n'a pas incrémenté son compteur. $conf_2$ est donc une configuration nominale initiale.

Une preuve plus élaborée montre que le compteur peut être incrémenté modulo `N-1`.

### 4.4 Extension de l'algorithme

Nous allons maintenant lever l'abstraction qui consiste à supposer qu'un site peut lire le compteur de son prédécesseur. $cprec_i$ sera maintenant une copie distante de ce compteur maintenu par l'algorithme de mémoire virtuelle répartie. Cependant, nous devons prendre quelques précautions quant à la valeur du modulo. Pour l'instant nous le posons égal à `H` et nous discuterons ensuite de sa valeur précise. Nous donnons en totalité l'algorithme composé pour illustrer les précautions à prendre lors de la composition.

Variables du site i

Tout d'abord chaque site joue à la fois le rôle du propriétaire pour son compteur et celui du lecteur pour le compteur précédent. Ceci explique que la variable $h_i$ soit "dédoublée".

- $c_i$ : compteur du site i.
- $cprec_i$ : copie distante du compteur du site précédent i sur l'anneau.
- $f_i$ : fonction à exécuter en section critique
- $h_i$ : compteur associé à $c_i$.
- $état_i$ : état du service de lecture. Cette variable prend une valeur parmi (repos, prem_att, sec_att).
- $hprec_i$ : compteur associé à $cprec_i$
- $val_i$ : valeur à renvoyer à la lecture
- $time_out_i$ : constante contenant la valeur d'un délai de suspension.
- $suivant_i$ : constante contenant l'identité du suivant sur l'anneau.

Algorithme du site i

La première partie de l'algorithme est presque identique à l'algorithme précédent à la différence près que $cprec_i$ donne lieu à une lecture distante.

```
circuler()
Début
     Tant que VRAI faire
          // le read introduit une temporisation implicite
          cpreci=read();
          Si   (i==i0)&&(ci==cpreci)||
               (i!=i0)&&(ci!=cpreci) Alors
               fi();
               fi=NULL;
               Si (i==i0) Alors
                    ci = ci+1 % H;
               Sinon
                    ci = cpreci;
               Finsi
          Finsi
     Fin tant que
Fin
```

```
SC(f)
Début
     fi=f;
     Attendre(fi==NULL);
Fin
```

La deuxième partie consiste à faire jouer les deux rôles de la lecture distante à un même site selon la variable.

La fonction `read()` est inchangée.

```
read()
Début
     étati=prem_att;
     Attendre(étati==repos);
     renvoyer(vali);
Fin
```

Ce processus utilitaire doit être ordonnancé de manière (faiblement) équitable avec le processus `circuler()`. Autrement dit, aucun des deux ne doit être privé indéfiniment d'exécution.

```
régénérer()
Début
     Tant que VRAI faire
          envoyer_à(suivanti,(ci,hi));
          Armer(time_outi);
          Attendre();
     Fin tant que
Fin
```

Cette primitive correspond au lecteur distant. Aussi elle manipule $hprec_i$.

```
sur_réception_de(j,(v,h))
Début
     Si (hpreci!=h) Alors
          hpreci=h;
          Si (étati == prem_att) Alors
               étati=sec_att;
          Sinon si (étati == sec_att) Alors
               vali=v;
               étati=repos;
          Finsi
     Finsi
     envoyer_à(j,(acq,h));
Fin
```

Cette primitive correspond au propriétaire. Aussi elle manipule $h_i$.

```
sur_réception_de(j,(acq,h))
Début
     Si (hi==h) Alors
               hi = hi+1 %B;
               envoyer_à(suivanti,(ci,hi));
               Armer(time_outi);
     Finsi
Fin
```

### *4.5 Correction de l'algorithme étendu*

Nous remarquons d'abord que les différents protocoles de lecture distante (un par site) sont indifférents à l'exécution de la partie liée à la section critique. Nous avons déjà établi que ce protocole est auto-stabilisant. Nous nous plaçons donc en un instant qui suit la stabilisation de ces algorithmes.

Dans ce cas, nous pouvons identifier l'instruction "$cprec_i$`=read();`" à l'instruction "$cprec_i = c_j$" d'une exécution séquentielle de l'algorithme pour `j` prédecesseur de `i` sur l'anneau.

Posons `H=2.N+1` et démontrons la stabilisation.

Nous procédons en plusieurs étapes.

1. Le compteur du site distingué ne reste jamais bloqué. Supposons qu'il n'en soit pas ainsi. Alors le site suivant ne peut plus faire que deux mises à jour de sa variable : une avant la copie du compteur précédent, une après. De même, le suivant de ce site ne peut faire que quatre mises à jour, etc. On obtient donc une configuration où aucun site ne peut faire de mise à jour et de là une autre configuration où toutes les copies distantes sont égales aux variables. En raisonnant successivement sur les sites non distingués, on conclut que toutes les valeurs sont identiques. Mais dans ce cas, le site distingué peut faire une mise à jour, d'où la contradiction.
2. Il y a moins de `2.N+1` valeurs différentes dans la configuration initiale. Donc une valeur de compteur `val` est absente. D'après le premier point, on atteint une configuration $conf_1$ où le compteur du site distingué a cette valeur (ce peut être la configuration initiale).
3. Examinons maintenant la configuration $conf_2$ qui suit $conf_1$ dans laquelle le site distingué peut à nouveau modifier son compteur. Puisque `val` est absente des compteurs des sites non distingués et de toutes les copies, un raisonnement par récurrence (en "remontant" l'anneau à partir du site distingué) montre que chacun d'entre eux doit positionner au moins une fois son compteur à `val`. Un autre raisonnement par récurrence (cette fois-ci dans le sens de l'anneau) montre que ces sites ne peuvent ensuite le modifier tant que le site distingué n'a pas incrémenté son compteur. $conf_2$ est donc une configuration nominale initiale.

Une preuve plus élaborée montre que le compteur peut être incrémenté modulo `2.N-1`.

Remarque Ce type de composition est pratiqué de manière très générale pour obtenir des algorithmes auto-stabilisants traitant des tâches complexes.

## 5 Exercices

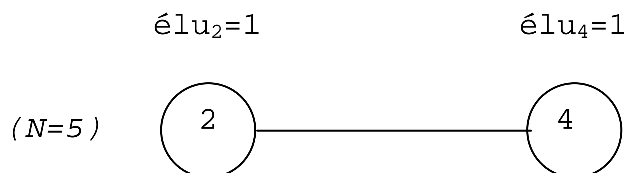
***Sujet 1***

On se propose de définir un algorithme d'élection auto-stabilisante sur un graphe de communication quelconque. Le site élu sera le site de plus petite identité. Nous précisons d'abord les hypothèses sur l'environnement :

- Chaque site `i` a en mémoire permanente sa propre identité et l'ensemble des identités de ses voisins stocké dans la constante $Voisins_i$.
- Chaque site peut lire les variables de ses voisins.
- Les identités des sites du réseau forment un sous-ensemble de l'intervalle `0..N`.
- Chaque site ne connaît que l'identité de ses voisins.
- Le site maintient une variable $élu_i$ à valeurs dans `0..N` qui contiendra l'identité du site élu.

Une première tentative consiste pour le site `i` à mettre à jour sa variable $élu_i$ périodiquement avec le minimum de `i` et des variables $élu_j$ pour tout $j \in Voisins_i$.

**Question 2** Expliquer en vous appuyant sur la figure ci-dessous qui représente un état initial possible pourquoi cet algorithme ne fonctionne pas.



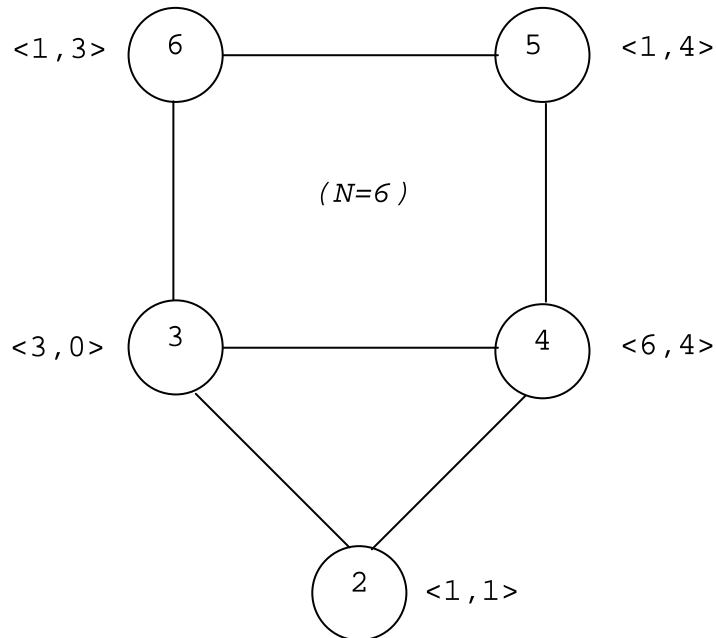

Afin de remédier au problème découvert, on décide d'ajouter une nouvelle variable $distance_i$ qui représente la distance minimale supposée à laquelle se trouve l'élu. $distance_i$ prend ses valeurs dans `0..N`. On modifie l'algorithme précédent avec les règles suivantes :

- La variable $élu_i$ est mise à jour périodiquement avec le minimum de `i` et des variables $élu_j$ pour tout $j \in Voisins_i$ tel que $distance_j < N$.
- Puis si $élu_i=i$ alors $distance_i$ est mise à jour avec `0` sinon $distance_i$ est mise à jour avec le minimum des $distance_j+1$ tel que $élu_j=élu_i$.

**Question 3** Ecrire l'algorithme du site `i`.

**Question 4** Dérouler l'algorithme à partir de l'état initial décrit à la figure ci-dessous en exécutant l'algorithme par ronde. Au cours d'une ronde, les sites 2, 3, 4, 5 et 6 exécutent à tour de rôle leur mise à jour. On dessinera une figure par état obtenu à l'issue d'une ronde jusqu'à stabilisation de l'algorithme.

Dans le couple `<x,y>`
`x` désigne $élu_i$
et `y` désigne $distance_i$




**Question 5** Etablir la correction de l'algorithme en décomposant toute exécution en rondes telles qu'au cours d'une ronde, tout site a effectué au moins une mise à jour de ses variables. Donner une borne, fonction de `N`, sur le nombre maximal de rondes qu'il faut pour que toutes les variables $élu_i$ contiennent la plus petite identité du réseau.

## 6 Références

[Dij73] E.W. Dijkstra "Self-stabilization in spite of distributed control" EWD391 Springer-Verlag (1973) pp 41-46.

[Dol89] S. Dolev, A. Israeli, S. Moran "Self-stabilization of dynamic systems" Proceedings of the MCC Workshop on Self-stabilizing systems, MCC Technical Report N° STP-379-89 (1989)

[Dol97] S. Dolev, A. Israeli, S. Moran "Resource bounds for self-stabilizing message driven protocols" SIAM Journal of Computing 26, pp 273-290 (1997)

[Dol00] S. Dolev "Self-Stabilization" MIT press (2000)

[Gou91] M.G. Gouda, N. Multari "Stabilizing communication protocols" IEEE Transactions on computers, 40, pp 448-458 (1991)