CHAPITRE VI

L'ELECTION

version du 16 janvier 2004

1 Problématique

Il arrive que même si les stations d'une application répartie ont des fonctionnalités similaires, certaines tâches spécifiques doivent être réalisées par une station unique. Aussi avant de démarrer l'application, les sites doivent se mettre d'accord sur l'identité de cette station. Cette opération est communément appelée élection et fait l'objet du présent chapitre.

Donnons deux exemples de l'intérêts de l'élection :

- Dans une application de type maître-esclaves, tous les serveurs sont redondants pour le traitement des requêtes de lecture des clients. Cependant les requêtes d'écriture ne sont traitées que par un seul serveur (le maître) qui transmet ensuite les modifications aux autres sites (les esclaves). On pourra, par exemple, imaginer une gestion de mots de passe qui permet d'ouvrir une session sans passer par le réseau.
- Certaines applications nécessitent une phase d'initialisation exécutée par un seul site. L'exemple le plus simple est celui de la circulation d'un jeton unique entre les sites. Dans ce cas l'initiateur est le premier possesseur du jeton.

Il est naturellement possible de fixer l'identité de ce site dans le code ou mieux dans un fichier de configuration. Cependant une telle solution présente des inconvénients. En cas de changement de configuration du réseau local, l'administrateur doit modifier le (ou les) fichier(s) de configuration. Un arrêt temporaire mais durable de la machine choisie nécessite une intervention manuelle.

Aussi, nous présentons ici des solutions qui sont dynamiques au sens où l'élection a lieu uniquement lorsqu'un site de l'application a besoin d'utiliser cette identité. Autrement dit, le service que nous réaliserons a une interface constituée d'une seule fonction `leader()` qui renvoie l'identité du site élu. La contrainte à vérifier est que l'identité renvoyée soit toujours la même quelque soit l'instant ou le lieu de l'appel.

La suite du chapitre est organisée ainsi. Nous développons trois algorithmes selon le type de graphe de communication : anneau unidirectionel, anneau bidirectionnel et graphe quelconque. Ces trois algorithmes ont de plus l'avantage de ne nécessiter de chaque site que la connaissance de ses voisins dans le graphe. Autrement dit, l'insertion d'un nouveau site ne requiert aucune intervention au niveau des sites qui ne sont pas ses voisins (moyennant le fait que le type de graphe est inchangé dans le cas de graphes spécifiques).

2 Algorithme de Chang et Roberts [Cha79]

2.1 Principe et réalisation de l'algorithme

Cet algorithme s'applique à un anneau unidirectionnel. Chaque site n'émet des messages que vers le site suivant sur l'anneau.

Lorsque l'application « réclame » l'identité du leader, trois cas se présentent :

- Le processus d'élection est terminé et le service renvoie immédiatement l'identité du site élu.
- Le processus d'élection est en cours et connu du service. Dans ce cas, le service attend la terminaison du processus afin de renvoyer l'identité du leader.
- Le service n'est pas au courant d'un processus d'élection engagé. Dans ce cas, il envoie une requête circuler sur l'anneau afin de devenir le leader. Si la requête lui revient, il conclut que son offre est acceptée et envoie un message de confirmation circuler sur l'anneau pour indiquer aux autres sites qu'il est le leader.

Lorsqu'une requête parvient à un autre site, deux cas se présentent :

- Le service du site était au repos ou avait provisoirement choisi un leader d'identité supérieure; il adopte l'initiateur de la requête comme nouveau leader potentiel et retransmet la requête au site suivant de l'anneau.
- Le service du site avait provisoirement choisi un leader d'identité inférieure; il ne donne pas suite à la requête.

La figure 7.1 décrit les phases successives d'une exécution possible de l'algorithme. Sur la figure 7.1.a, l'application du site 4 a appelé la fonction `leader()`ce qui a provoqué l'envoi d'une requête du service pour devenir « leader ». Sur la figure 7.1.b, trois autres applications ont appelé la fonction `leader()` : les sites 2,3,5. Puisqu'une requête a déjà été retransmise par le service du site 5, celui-ci n'envoie pas de requête et attend la fin de l'élection (il a adopté provisoirement 4 comme leader). Les deux autres sites envoient des requêtes. Sur la figure 7.1.c, seules deux requêtes circulent encore puisque la requête du site 4 a été « détruite » par le site 2. Tous les sites sont maintenant au courant du processus d'élection. Sur la figure 7.1.d, seule la requête du site 2 circule encore car celle du site 3 a aussi été « détruite » par le site 2. Notez que tous les sites ont adopté provisoirement le site 2 comme leader. Sur la figure 7.1.e, le site 2 sait qu'il est élu puisque sa requête lui est revenue. Il envoie un message de confirmation aux autres sites. Sur la figure 7.1.f, tous les sites ont adopté définitivement le site 2 comme leader et le message de confirmation sera détruit à la réception par le site 2.

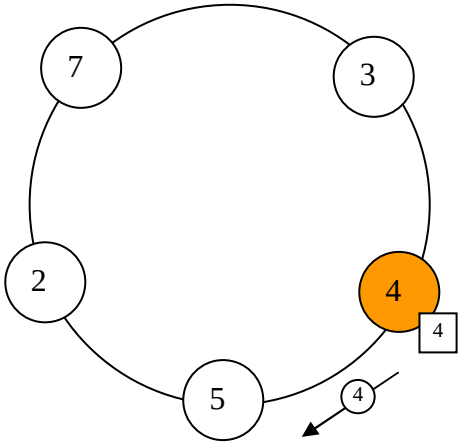


Figure 7.1.a

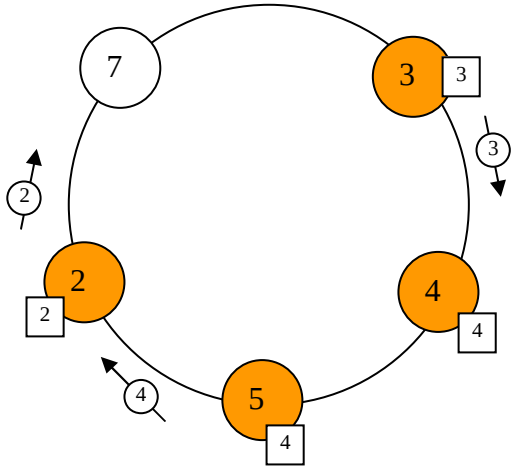


Figure 7.1.b

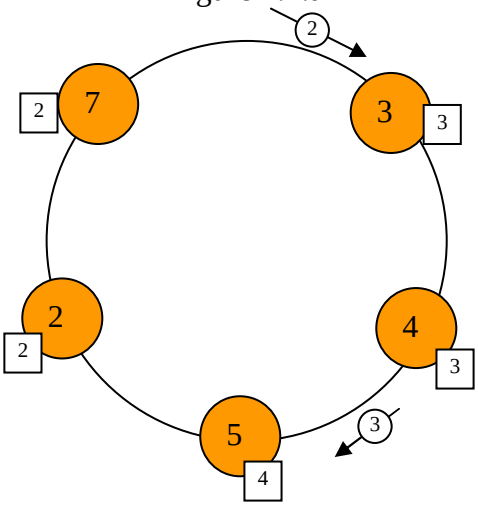


Figure 7.1.c

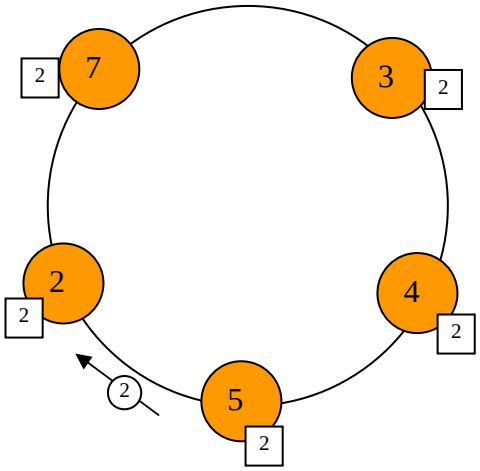


Figure 7.1.d

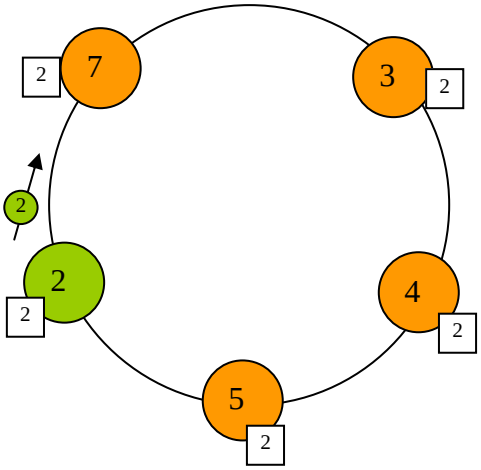


Figure 7.1.e

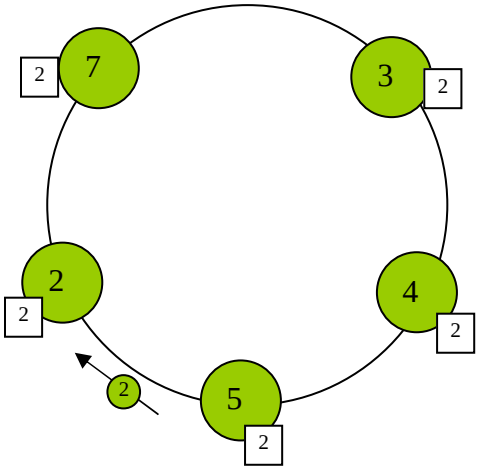


Figure 7.1.f

Nous sommes maintenant en mesure d'énoncer l'algorithme.

Variables du site `i`

- $\texttt{suivant}_i$: constante contenant l'identité du site successeur de `i` sur l'anneau.
- $\texttt{état}_i$: état du service. Cette variable prend une valeur parmi l'ensemble des valeurs (`repos`,`en_cours`,`terminé`). Cette variable est initialisée à `repos`.
- $\texttt{chef}_i$: identité du site élu.

Algorithme du site `i`

Si aucun processus d'élection n'a atteint le site `i`, le service initialise un tel processus. Dans tous les cas, il attend que le processus d'élection soit terminé pour renvoyer l'identité du site élu.

```
leader()
Début
     Si (état_i==repos) Alors
          état_i=en_cours;
          chef_i=i;
          envoyer_à(suivant_i,(req,i));
     Finsi
     Attendre(état_i==terminé);
     renvoyer(chef_i);
Fin
```

Si le site est au repos, la réception d'une requête provoque le changement d'état et la retransmission de la requête. Dans le cas où le processus reçoit une "meilleure" requête, il en tient compte et retransmet aussi la requête. Enfin si une requête parvient à son initiateur, ce site est élu et il avertit les autres sites.

```
sur_réception_de(j,(req,k))
Début
     Si (état_i==repos || k<chef_i) Alors
          état_i=en_cours;
          chef_i=k;
          envoyer_à(suivant_i,(req,k));
     Sinon si (i==k) Alors
          état_i=terminé;
          envoyer_à(suivant_i,(conf,i));
     Finsi
Fin
```

La confirmation du site élu fait un tour de l'anneau.

```
sur_réception_de(j,(conf,k))
Début
     Si (i!=k) Alors
          envoyer_à(suivant_i,(conf,k));
          état_i=terminé;
     Finsi
Fin
```

2.2 Preuve de l'algorithme

Un site dont l'application appelle `leader()` alors que son service est au repos sera désigné comme un initiateur. Soit i_0 l'initiateur d'identité minimale, la requête (req, i_0) circule de bout en bout et atteint i_0. Le site i_0 passera donc à l'état `terminé`, enverra son message de confirmation qui entrainera à sa réception le passage de tous les autres sites à ce même état et l'adoption de i_0 comme leader.

Il reste à démontrer qu'aucun autre message de confirmation ne circulera sur le réseau. Soit i_1 un autre initiateur, si la requête de i_1 parvient à i_0, elle sera détruite puisque l'émission de la requête de i_0 précède cette réception (par définition d'un initiateur). En conséquence, aucun autre message de confirmation ne sera émis.

2.3 Complexité de l'algorithme

2.3.1 Complexité au pire des cas

Nous désirons obtenir un ordre de grandeur sur `Pire(n)`, le nombre maximum de messages échangés durant une élection sur un anneau composé de `n` sites. Notons d'abord que chaque site envoie au plus une requête. Chaque requête donne lieu à au plus `n` envois de message. Enfin, le message de confirmation donne lieu à exactement `n` envois de message. En cumulant ces bornes nous obtenons :

$$Pire(n) \leq n^2 + n$$

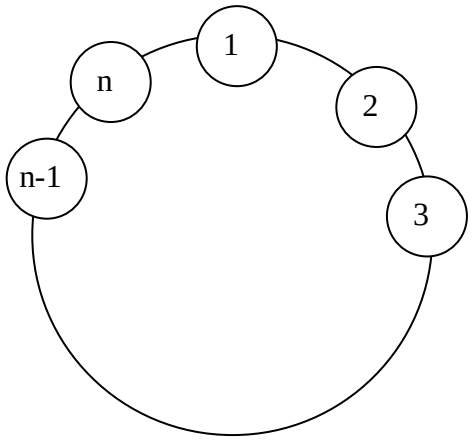


Figure 7.2 : Une configuration défavorable

Examinons le cas de la figure 7.2. où tous les sites sont initiateurs, la requête issue de i est envoyée exactement `n-i+1` fois. Ce qui nous donne pour cette exécution particulière un nombre de messages égal à :

$$\sum_{i=1 \text{ à } n} (n-i+1) + n = n(n+1)/2 + n \leq Pire(n)$$

Cet encadrement nous fournit l'ordre de grandeur recherché :

$$Pire(n) = \theta(n^2).$$

2.3.2 Complexité en moyenne

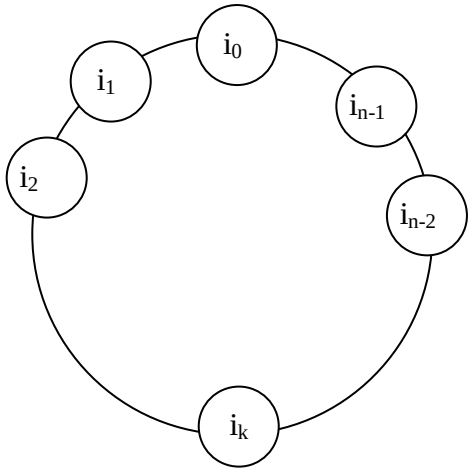


Figure 7.3 : Une configuration aléatoire

Nous cherchons maintenant à évaluer $Moyenne(n)$ qui représente le nombre moyen de messages échangés sur l'ensemble des exécutions possibles (nous supposons dans la suite que tous les sites sont initiateurs). Posons i_0 le site d'identité minimale et considérons la configuration aléatoire de la figure 7.3. Soit X_k la variable aléatoire représentant le nombre de messages issus de la requête de i_k (pour $k \neq 0$). Puisque cette requête ne pourra être relayée par i_0, nous avons : $Prob(X_k \geq k+1)=0$.

Pour que cette requête soit relayée au moins t fois pour $t \leq k$ il faut et il suffit que i_k soit l'identité minimale dans l'ensemble des identités $\{i_k,...,i_{k-t+1}\}$. Nous supposons que toutes les configurations sont équiprobables, donc cette minimalité est satisfaite avec une probabilité $1/t$. Autrement dit, $Prob(X_k \geq t)= 1/t$.

Nous appliquons maintenant un résultat élémentaire sur l'espérance d'une variable aléatoire à valeurs entières :

$E(X) =_{def} \sum_{t=1 \text{ à } \infty} t.Prob(X=t)$
$= \sum_{t=1 \text{ à } \infty} \sum_{s=1 \text{ à } t} Prob(X=t)$ *(remplacement du produit par la somme)*
$= \sum_{s=1 \text{ à } \infty} \sum_{t=s \text{ à } \infty} Prob(X=t)$ *(inversion des sommes)*
$= \sum_{s=1 \text{ à } \infty} Prob(X \geq s)$

D'où : $E(X_k) = \sum_{t=1 \text{ à } k} 1/t$

Nous sommons ces différentes espérances en ajoutant les deux tours (requête et confimation) dûs à i_0 :

$$\text{Moyenne}(n) = \sum_{k=1 \text{ à } n-1} \sum_{t=1 \text{ à } k} 1/t + 2.n$$
$$= \sum_{t=1 \text{ à } n-1} \sum_{k=t \text{ à } n-1} 1/t + 2.n$$
$$= \sum_{t=1 \text{ à } n-1} (n-t)/t + 2.n$$

(inversion des sommes et remplacement de la somme par le produit)

$$= \sum_{t=1 \text{ à } n-1} n/t + \sum_{t=1 \text{ à } n-1} -t/t + 2.n$$
$$= n.\sum_{t=1 \text{ à } n-1} 1/t + n + 1$$
$$= n.\sum_{t=1 \text{ à } n-1} 1/t + n + n.(1/n)$$

Finalement :

$$\text{Moyenne}(n) = n.\sum_{t=1 \text{ à } n} 1/t + n$$

Il nous reste à obtenir un ordre de grandeur de cette quantité. Nous nous servons de l'encadrement immédiat :

$$\int_t^{t+1} 1/x \, dx \leq 1/t \leq \int_{t-1}^{t} 1/x \, dx \text{ pour } t \geq 2$$

ce qui nous donne par sommation :

$$\log(n+1) = \int_1^{n+1} 1/x \, dx \leq \sum_{t=1 \text{ à } n} 1/t \leq 1 + \int_1^{n} 1/x \, dx = 1+\log(n)$$

D'où :

$$\text{Moyenne}(n) = \theta(n.\log(n))$$

3 Algorithme de Franklin [Fra82]

3.1 Principe et évaluation de l'algorithme

Cet algorithme s'applique à un anneau bidirectionnel. Il se décompose en "tours" où les compétiteurs sont les initiateurs (au même sens que dans l'algorithme précédent). A chaque tour, un initiateur survivant envoie à ses initiateurs voisins (à gauche et à droite) sa candidature. De ce fait, chaque initiateur reçoit les requêtes des initiateurs voisins gauche et droite et ne survit au tour suivant que s'il a la plus petite identité. Le tournoi s'achève quand un initateur sait qu'il est ou sera au prochain tour le seul survivant. Il acquiert cette certitude soit en recevant l'un des messages qu'il a envoyé (ou éventuellement les deux) soit en recevant deux messages d'un même site. Les sites "éliminés" participent à l'algorithme en transmettant les messages qu'ils reçoivent. Comme dans l'algorithme précédent un message de confirmation achève l'algorithme.

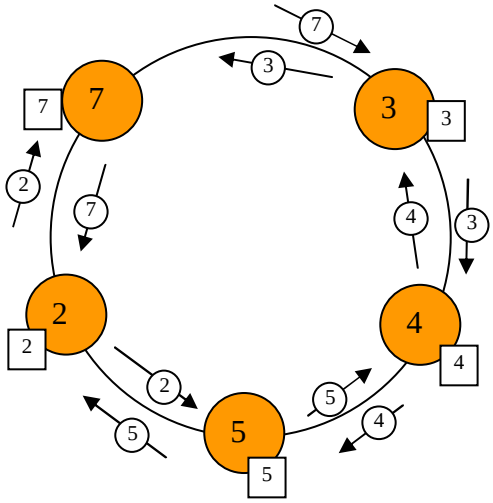


Figure 7.5.a

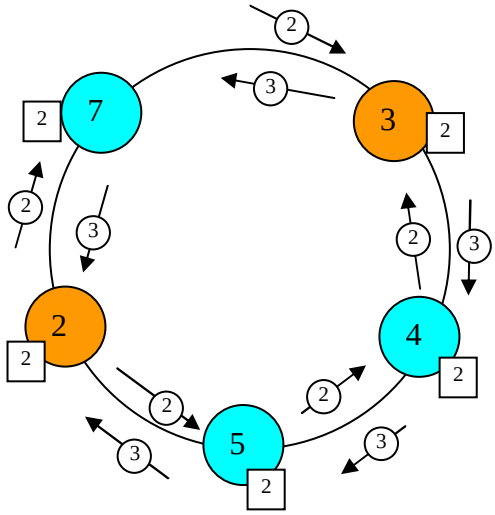


Figure 7.5.b

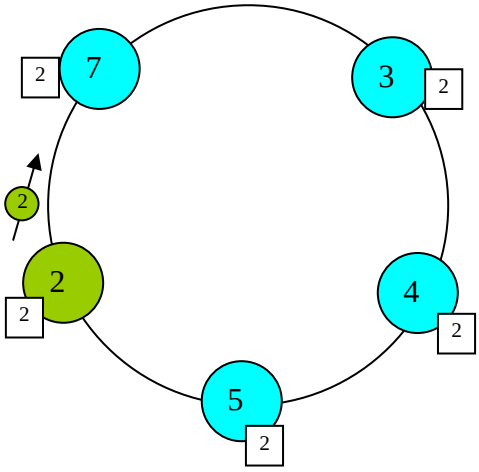


Figure 7.5.c

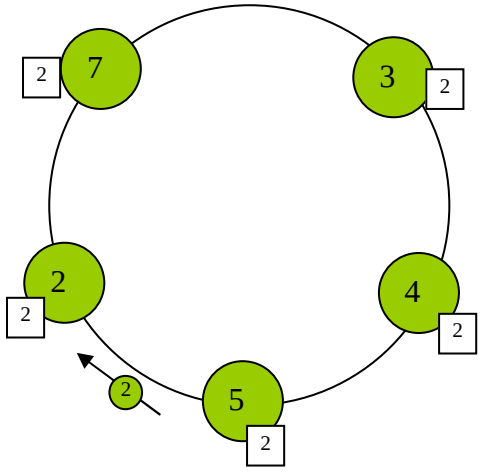


Figure 7.5.d

Nous avons représenté sur la figure 7.5, une exécution possible de l'algorithme. Sur la figure 7.5.a, toutes les applications ont appelé la fonction `leader()`. Par conséquent, chaque site envoie sa candidature à gauche et à droite. Seuls le site 2 et 3 survivent et se retrouvent au deuxième tour. Aussi, ils s'envoient à nouveau leur candidature qui sont relayés par les sites éliminés (figure 7.5.b). Lors de ce tour le site 2 est le seul survivant et de plus il le sait car il a reçu deux messages issus du site 3. Il envoie donc un message de confirmation (figure 7.5.c). Lorsque ce message de confirmation a fait le tour de l'anneau (figure 7.5.d), le processus d'élection est terminé.

Evaluons cet algorithme dans le pire des cas. Nous remarquons d'abord qu'à chaque tour exactement $2.n$ messages sont échangés (un tour dans chaque sens). Nous majorons le nombre de tours de la manière suivante. A chaque initiateur survivant d'un tour on associe son voisin initiateur de droite éliminé lors de ce tour. Cette fonction est injective. Donc le nombre d'éliminés durant un tour est au moins égal au nombre de survivants. Autrement dit, le nombre de survivants est au moins divisé par 2 à chaque tour. Sachant qu'il peut y avoir un tour supplémentaire (inutile), on obtient que le nombre de tours est majoré par $ENT(\log_2(n))+1$.

En ajoutant le message de confirmation, on obtient :

$$Pire(n) \leq 2.n.(ENT(\log_2(n))+1)+ n = \theta(n.\log(n))$$

Nous laissons le soin au lecteur d'exhiber un exemple d'exécution démontrant que l'on a :

$$Pire(n) = \theta(n.\log(n))$$

Indication On placera $n=2^k$ sites de la manière itérative suivante. On place d'abord les sites 1 et 2. Une fois placés les 2^i sites de plus basses identités, on intercale les 2^i sites suivants entre les sites déjà placés. On vérifie par récurrence que k tours sont nécessaires.

Notons que cet algorithme a été adapté pour fonctionner sur un anneau unidirectionnel avec la même complexité de messages [Pet82] et [Dol82].

3.2 Description

Il y a une difficulté supplémentaire due à l'asynchronisme du réseau. Un site peut être en avance d'un tour sur un candidat voisin. Par exemple si i_0, i_1, i_2, i_3 et i_4 sont des candidats "voisins" à un tour donné, i_3 peut avoir éliminé i_2 et i_4 et avoir envoyé ses messages du tour suivant alors que i_1 attend encore la réponse de i_0. Il faut donc que i_1 conserve ce message en avance pour le traiter au tour suivant. Notons qu'un site ne peut recevoir qu'un message en avance et qu'il le reçoit dans la même direction que le premier message reçu.

Variables du site `i`

- $suivant_i$: constante contenant l'identité du site successeur de `i` sur l'anneau.
- $précédent_i$: constante contenant l'identité du site prédécesseur de `i` sur l'anneau.
- $état_i$: état du service. Cette variable prend une valeur parmi (`repos`, `en_cours`, `attente`, `terminé`). Cette variable est initialisée à `repos`.
- $nbreq_i$: nombre de requêtes reçues au cours d'un tour

- $conc_i$: identité du site dont on reçoit la première des deux requêtes du tour courant. Lorsqu'elle vaut `i`, cela signifie qu'aucune requête en avance n'est reçue.
- dir_i : booléen indiquant la direction d'où vient la première des deux requêtes du tour courant
- $concav_i$: identité du site dont on reçoit la première des deux requêtes du tour suivant
- $chef_i$: identité du site (provisoirement) élu.

Algorithme du site `i`

Si aucun processus d'élection n'a pas atteint le site `i`, le service initialise un tel processus. Au cours de ce processus, il propose sa candidature dans les deux sens de l'anneau tant qu'il n'est pas éliminé ou qu'il ne sait pas qu'il est le seul survivant. Dans tous les cas, il attend que le processus d'élection soit terminé pour renvoyer l'identité du site élu. La boucle `répéter` correspond à la participation du site aux tours successifs. A chaque changement de tour, il faut prendre garde à traiter l'éventuelle requête en avance.

```
leader()
Début
     Si (état_i==repos) Alors
          état_i=en_cours;
          chef_i=i;concav_i=i;
          Répéter
               nbreq_i=0;
               Si concav_i!=i Alors
                    nbreq_i=1;
                    conc_i=concav_i;
                    Si (conc_i<chef_i) Alors chef_i=conc_i; Finsi
                    concav_i=i;
               Finsi
               envoyer_à(suivant_i,(req,i));
               envoyer_à(précédent_i,(req,i));
               Attendre(nbreq_i==2);
          Jusqu'à(état_i !=en_cours);
          Si concav_i!=i Alors
               Si (dir_i) Alors
                    envoyer_à(suivant_i,(req,concav_i));
               Sinon
                    envoyer_à(précédent_i,(req,concav_i));
               Finsi
          Finsi
     Finsi
     Attendre(état_i==terminé);
     renvoyer(chef_i);
Fin
```

A la réception d'une requête, on met à jour si nécessaire l'identité provisoire du leader. Dans le cas où on est un initiateur survivant (état à `en_cours`), on enregistre les deux requêtes voisines en prenant garde à mémoriser une requête en avance. Sur la réception de la deuxième, on teste si on a survécu puis si on est le seul survivant. Dans le cas où on est en `attente`, on retransmet les messages (fonction proxy).

```
sur_réception_de(j,(req,k))
Début
     Si (état_i == repos || k<chef_i) Alors
          chef_i = k;
     Finsi
     Si (état_i == en_cours) Alors
          Si (nbreq_i==0) Alors
               conc_i=k;
               nbreq_i=1;
               dir_i=(j==précédent_i);
          Sinon si ((dir_i&&j==précédent_i)||
          (!dir_i&&j!=précédent_i)) Alors
               concav_i=k;
          Sinon
               nbreq_i=2;
               Si (chef_i<i) Alors
                    état_i=attente;
               Sinon si (conc_i==i||k==conc_i) Alors
                    état_i = terminé;
                    envoyer_à(suivant_i,(conf,i));
               Finsi
          Finsi
     Sinon
          état_i=attente;
          Si (j==précédent_i) Alors
               envoyer_à(suivant_i,(req,k));
          Sinon
               envoyer_à(précédent_i,(req,k));
          Finsi
     Finsi
Fin
```

La confirmation du site élu fait un tour de l'anneau.

```
sur_réception_de(j,(conf,k))
Début
     Si (i!=k) Alors
          envoyer_à(suivant_i,(conf,k));
          état_i=terminé;
     Finsi
Fin
```

4 Généralisation de l'algorithme de Chang et Roberts

4.1 Algorithme de parcours du graphe de communication [Tar1895]

4.1.1 Présentation informelle et preuve

On désire généraliser l'algorithme de Chang et Roberts à un graphe de communication quelconque. La difficulté de cette généralisation réside dans le fait que la connaissance du graphe par un site se limite à ses voisins. Nous allons maintenant établir que cette connaissance minimale est suffisante pour gérer un parcours du graphe de communication qui vérifie les propriétés suivantes :

1. Ce parcours se termine chez l'initiateur.
2. Lorsqu'il se termine, ce parcours a visité au moins une fois tous les sites et a traversé exactement une fois chaque canal de communication dans les deux directions.

Attention, ***le parcours dépend de l'initiateur*** mais ceci n'affecte pas la correction de l'algorithme d'élection dont nous verrons la preuve plus loin.

Nous allons expliquer l'algorithme sur un exemple décrit à la figure 7.6.

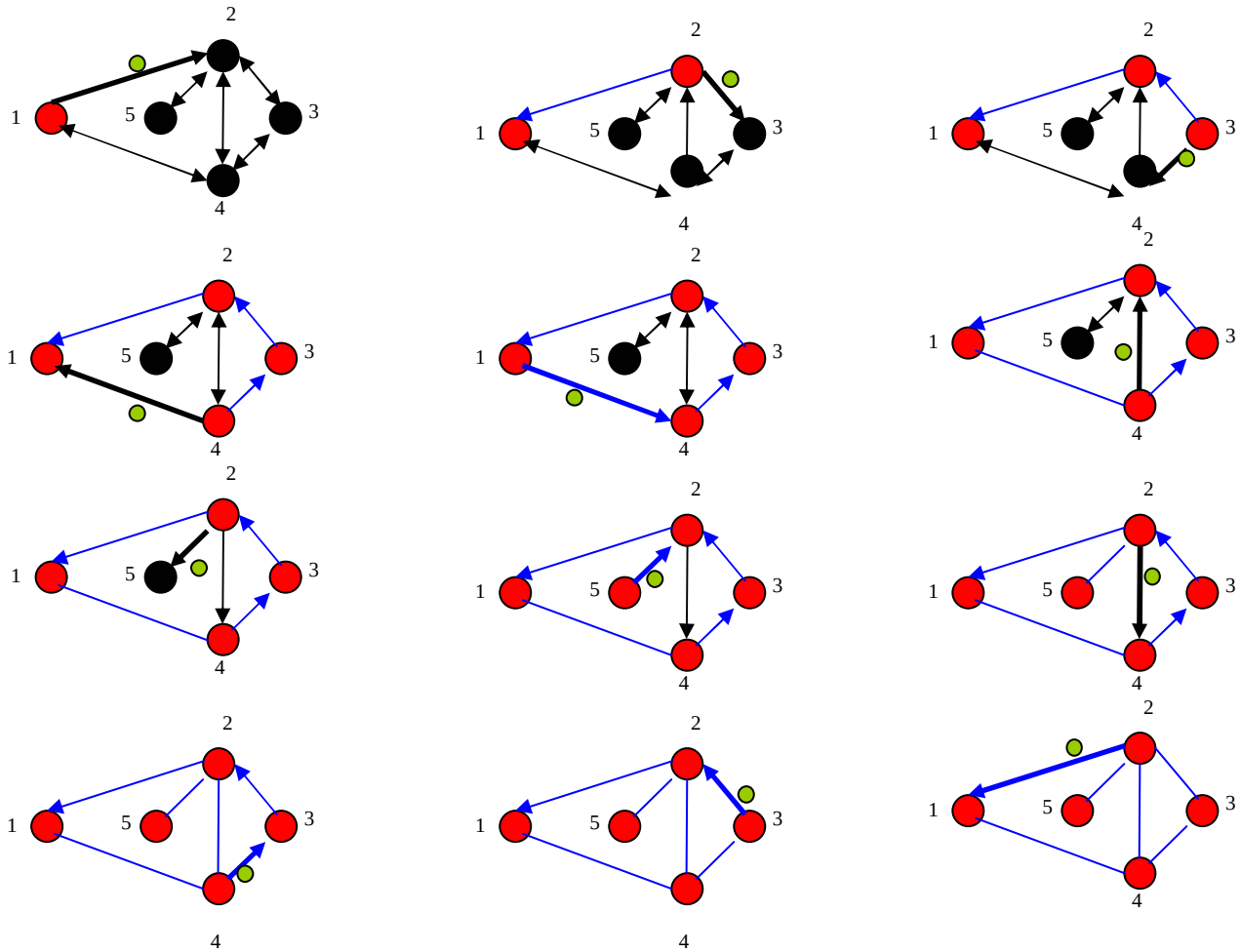


Figure 7.6 Un parcours de graphe

Chaque canal sortant est considéré comme initialement "ouvert" (ce qui est représenté sur la figure par une flèche à l'extrémité concernée). Lorsque le message associé au parcours est envoyé sur un canal ouvert, ce canal se "ferme" et ne pourra plus être utilisé par le parcours. Un site déjà visité par ce message est indiqué sur la figure en rouge et dans le cas contraire en noir. De plus, lors de la première visite par le message (excepté pour l'initiateur), le site mémorise le canal comme étant celui de son père, indiqué en bleu sur la figure. Il n'utilisera ce canal qu'après avoir utilisé tous les autres canaux.

Déroulons maintenant l'exemple sur les 12 schémas successifs de la figure :

1. Le site initie un parcours en envoyant le message au site 2. Par conséquent, il clôt ce canal.
2. A la réception, le site 2 mémorise que le canal de 2 vers 1 est le canal de "retour". Il a le choix entre les trois autres canaux et décide de l'envoyer au site 3 (canal fermé).
3. A la réception, le site 3 mémorise que le canal de 3 vers 2 est le canal de "retour". Il ne dispose que d'un autre canal et l'envoie au site 4 (canal fermé).
4. A la réception, le site 4 mémorise que le canal de 4 vers 3 est le canal de "retour". Il a le choix entre les deux autres canaux et décide de l'envoyer au site 1 (canal fermé).
5. A la réception, le site 1 ne dispose plus que d'un autre canal et le renvoie au site 4 (canal fermé).
6. A la réception, le site 4 ne dispose plus que d'un canal différent du canal de retour et le renvoie au site 2 (canal fermé).
7. A la réception, le site 2 a le choix entre deux canaux différents du canal de retour et décide de l'envoyer au site 5 (canal fermé).
8. A la réception, le site 5 mémorise que le canal de 5 vers 2 est le canal de "retour". Il ne dispose d'aucun autre canal et le renvoie au site 2 (canal fermé).
9. A la réception, le site 2 ne dispose plus que d'un canal différent du canal de retour et le renvoie au site 4 (canal fermé).
10. A la réception, le site 4 ne dispose que du canal de retour et le renvoie au site 3 (canal fermé).
11. A la réception, le site 3 ne dispose que du canal de retour et le renvoie au site 2 (canal fermé)
12. A la réception, le site 2 ne dispose que du canal de retour et le renvoie au site 1 (canal fermé). Lorsque ce message arrive au site 1, celui-ci n'ayant plus de canal ouvert conclut que le parcours est terminé.

Avant de développer cet algorithme, nous établissons sa correction. Appelons pour un site i, le nombre de canaux sortants (et donc entrants) n_i.

Tout d'abord, le parcours se termine nécessairement car un canal n'est emprunté qu'au plus une fois et le nombre de canaux est fini. Supposons qu'il se termine ailleurs que chez l'initiateur. Ceci signifie que ce site i n'a plus de canal sortant ouvert, lors d'une visite. Or puisqu'il décrémente le nombre de canaux sortants à chaque visite. On en conclut qu'il a été visité plus de n_i fois. Mais puisque n_i est également le nombre de canaux entrants, cela signifie qu'un canal entrant a été emprunté plus d'une fois. Ce qui est impossible. Le parcours se termine donc chez l'initiateur.

Démontrons maintenant que lorsque l'algorithme se termine, alors tous les noeuds visités ont envoyé le message sur tous leurs canaux sortants. On le démontre par récurrence sur l'ordre des visites. Pour l'initiateur c'est évident, puisque c'est la condition d'arrêt de l'algorithme. Supposons par l'absurde que `i` soit le premier site visité à ne pas emprunter tous ses canaux sortants. Dans ce cas, il n'envoie pas le message à son "père", que nous notons `j`. Par hypothèse de récurrence, `j` a emprunté tous ses canaux sortants soit n_j canaux, mais il n'a pas été visité par la canal venant de i, donc il a été visité moins n_j de fois. Ce qui est contradictoire.

Démontrons finalement que tous les sites sont visités. Si ce n'est pas le cas, on partitionne l'ensemble des sites entre ceux qui sont visités et les autres. Puisque le graphe est connexe, il existe au moins une arête reliant un site visité à un site non visité. D'après le paragraphe précédent, ce canal a été emprunté d'où la contradiction.

4.1.2 Réalisation de l'algorithme

Tout d'abord, nous modifions légérement l'algorithme pour permettre plus plusieurs parcours successifs initiés par un même site. Notre algorithme se compose de deux fonctions `initier(type)` qui initie un parcours avec un certain type de message et `faire_suivre(j,type,k)` qui fait suivre un type de message venant de `j` dans un parcours initié par `k`. Cette fonction est appelée à la réception dudit message.

Pour gérer un parcours initié par un quelconque des sites, on utilise un tableau T_i indicé par les sites. Ceci peut sembler contradictoire avec l'hypothèse de minimalité sur la connaissance des sites, mais il est facile de remplacer T_i par une allocation dynamique au prix d'une programmation plus "lourde".

L'algorithme travaille avec des ensembles de canaux (ou voisins). La fonction `extraire(ensemble)` renvoie un élement de l'ensemble (s'il est non vide) et le retire de celui-ci. On s'autorise bien entendu à tester si un ensemble est vide.

Variables du site `i`

- $Voisins_i$: constante contenant l'ensemble des voisins de `i` dans le graphe de communication.
- T_i`[1..N]`: tableau dont la cellule `j` contribue au parcours d'un jeton initié par `j`. Chaque cellule a deux champs :
  - T_i`[j].voisins`
  - T_i`[j].père` initialisé à `i` qui signifie dans le cas où `j`$\neq$`i` que le parcours de `j` n'est pas encore parvenu à `i`.
- $prochain_i$: variable contenant l'identité d'un voisin de `i`.

Algorithme du site `i`

Pour initier un parcours du graphe, on initialise le champ voisins aux voisins du site et on extrait un voisin de cet ensemble pour débuter le parcours.

```
initier(type)
Début
     T_i[i].voisins = Voisins_i;
     prochain_i = extraire(T_i[i].voisins);
     envoyer_à(prochain_i,(type,i));
Fin

sur_réception_de(j,(type,k))
Début
     Si (!faire_suivre(j,type,k)) Alors
          afficher("parcours terminé");
     Finsi
Fin
```

S'il s'agit de la première visite de ce parcours, on initialise les champs père et voisins de la cellule concernée en extrayant le "père" des voisins. S'il reste des voisins, on extrait l'un d'entre eux pour continuer le parcours. Sinon si `i` n'est pas l'initiateur du parcours, on emprunte le canal de son père. Le parcours étant terminé sur ce site, on réinitialise le champ `père` pour un éventuel nouveau parcours. Si `i` est l'initiateur, la fonction renvoie l'indication que le parcours est terminé.

```
faire_suivre(j,type,k)
Début
     Si (k!=i) && (T_i[k].père==i) Alors
          T_i[k].père = j;
          T_i[k].voisins = Voisins_i \{j};
     Finsi
     Si T_i[k].voisins != ∅ Alors
          prochain_i = extraire(T_i[k].voisins);
          envoyer_à(prochain_i,(type,k));
          renvoyer(VRAI);
     Sinon si (k!=i) Alors
          envoyer_à(T_i[k].père,(type,k));
          T_i[k].père = i;
          renvoyer(VRAI);
     Sinon
          renvoyer(FAUX);
     Finsi
Fin
```

4.2 Description de l'algorithme d'élection

Variables du site `i`

Aux variables nécessaires au parcours, on ajoute les variables nécessaires à l'élection.

- $\texttt{état}_\texttt{i}$: état du service. Cette variable prend une valeur parmi l'ensemble des valeurs (`repos`,`en_cours`,`terminé`). Cette variable est initialisée à `repos`.
- $\texttt{chef}_\texttt{i}$: identité du site élu.

Algorithme du site `i`

Si aucun processus d'élection n'a atteint le site `i`, le service initialise un tel processus par un parcours du graphe avec une requête. Dans tous les cas, il attend que le processus d'élection soit terminé pour renvoyer l'identité du site élu.

```
leader()
Début
     Si (état_i==repos) Alors
          état_i=en_cours;
          chef_i=i;
          initier(req);
     Finsi
     Attendre(état_i==terminé);
     renvoyer(chef_i);
Fin
```

Si le site est au repos, la réception d'une requête provoque le changement d'état. Dans le cas où le processus reçoit une "meilleure" requête, il en tient aussi compte. Dans le cas contraire, le parcours est avorté (disparition implicite de la requête) et retransmet aussi la requête. Si le parcours n'est pas avorté alors à l'aide de la fonction `faire_suivre()`, soit on continue le parcours soit si le parcours est teminé (`i` est alors le site élu) on initie un parcours d'un confirmation.

```
sur_réception_de(j,(req,k))
Début
     Si (état_i==repos || k<=chef_i) Alors
          état_i=en_cours;
          chef_i=k;
          Si !faire_suivre(j,req,k) Alors
               état_i=terminé;
               initier(conf);
          Finsi
     Finsi
Fin
```

La confirmation du site élu fait un parcours du graphe de communication.

```
sur_réception_de(j,(conf,k))
Début
     état_i=terminé;
     faire_suivre(j,conf,k);
Fin
```

4.3 Correction et complexité de l'algorithme d'élection

Supposons que deux requêtes soient envoyées par deux initiateurs. Alors le parcours de l'initiateur de plus grande identité ne pourra se terminer car il visitera l'autre initiateur d'après la propriété d'exhaustivité du parcours.

Donc seul le message de confirmation de l'iniateur de plus petite identité est émis et parcourt le graphe.

Chaque initiateur initie un parcours de graphe et l'élu initie un deuxième parcours pour le message de confirmation. En notant E le nombre d'arêtes du graphe de communication, on obtient :

$$Pire(n) \leq (n+1).(2.E) = \theta(n.E)$$

L'exemple d'exécution de l'algorithme de Chang et Roberts établit que l'on a :

$$Pire(n) = \theta(n.E)$$

On peut adapter cet algorithme avec les idées de l'algorithme de Franklin afin d'obtenir un algorithme qui se comporte plus efficacement dans le cas où des parcours intelligents du graphe de communication sont possibles (e.g. grille, tore, etc.) [Kor90].

5 Exercices

Sujet 1

Dans ce problème, `n` désigne le nombre de sites et `E` le nombre d'arêtes du graphe de communication. On suppose que l'exécution d'instructions est instantanée (i.e. en un temps négligeable devant les délais de transit).

Question 1 En supposant qu'il y ait un seul initiateur et que la transmission de chaque message prenne exactement `1` unité de temps, calculez le temps nécessaire à l'élection effectuée par l'adaptation (vue en cours) de l'algorithme de Chang et Roberts pour un graphe de communication quelconque. En déduire l'inconvénient majeur de cet algorithme.

Afin de remédier à cet inconvénient, on se propose de définir un nouvel algorithme d'élection. Les variables de ce nouvel algorithme sont les suivantes :

- $\texttt{Voisins}_i$: constante contenant l'ensemble des voisins de `i` dans le graphe de communication ;
- $\texttt{état}_i$: état du service. Cette variable prend une valeur parmi l'ensemble des valeurs (`repos`,`en_cours`,`terminé`). Cette variable est initialisée à `repos` ;
- $\texttt{chef}_i$: identité du site élu ;
- $\texttt{père}_i$: identité du site à qui envoyer une « dernière » requête ;
- $\texttt{nbreq}_i$: nombre de requêtes à recevoir avant d'envoyer la dernière requête ;
- $\texttt{temp}_i$: variable temporaire.

Lorsque l'application « réclame » l'identité du leader, trois cas se présentent :

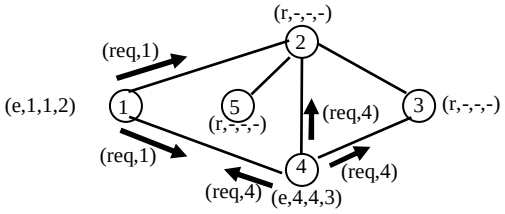
- Le processus d'élection est terminé et le service renvoie immédiatement l'identité du site élu.
- Le processus d'élection est en cours et connu du service. Dans ce cas, le service attend la terminaison du processus pour renvoyer l'identité du leader.
- Le service n'est pas au courant d'un processus d'élection engagé. Dans ce cas, il envoie à tous ses voisins une requête portant son identité afin de devenir le leader, se considère (provisoirement) comme l'élu et son propre père, initialise le nombre de requêtes attendues au nombre de voisins puis il attend la terminaison du processus pour renvoyer l'identité du leader.

Lorsqu'une requête parvient à un site, trois cas se présentent :

- Le service du site était au repos ou avait provisoirement choisi un leader d'identité supérieure. Il adopte l'initiateur de la requête comme nouveau leader potentiel et l'émetteur du message devient son nouveau père. S'il a des voisins différents de son père, il retransmet à tous ses voisins excepté son père la requête et (ré)initialise le nombre de requêtes attendues au nombre de voisins`-1`. Sinon il retransmet immédiatement la requête à son père.
- Le service du site avait provisoirement choisi un leader d'identité inférieure. Il ne donne pas suite à la requête.
- La requête correspond au leader actuel du site. Il décremente alors le nombre de requêtes attendues. Si celui-ci devient nul, soit le site n'est pas l'initiateur de la requête et la retransmet à son père, soit le site est l'initiateur, l'état du service est maintenant terminé et il envoie à ses voisins un message de confirmation.

La réception du message de confirmation est laissée à votre réflexion.

Question 2 Déroulez l'algorithme sur le graphe de communication ci-dessous en supposant que le site `1` et le site `4` appellent simultanément `leader(),` que la transmission de chaque message prenne exactement `1` unité de temps et que si deux messages arrivent simultanément sur un même site, on traite d'abord le message correspondant à l'émetteur de plus grande identité. Vous présenterez l'état de chaque site sous la forme $(\text{état}_i, \text{chef}_i, \text{père}_i, \text{nbreq}_i)$ et les messages qui circulent aux instants `0,1,2,...` jusqu'à la terminaison de l'algorithme. La situation initiale est décrite sur le graphe.



Question 3 Ecrivez les primitives suivantes :

- `leader()`
- `sur_réception_de(j,(req,k))`
- `sur_réception_de(j, conf)`

Question 4 Donnez une borne supérieure du nombre de messages échangés pour une élection en fonction de `n` et de `E`.

Question 5 On appelle distance entre deux sites, la longueur en nombre d'arêtes du plus court chemin entre deux sites. Le diamètre du graphe `D` est la plus grande distance entre deux sites quelconques. En supposant qu'une transmission de message prenne au plus `1` unité de temps, indiquez en fonction de `D`, le temps maximum entre le premier appel à `leader()` et le moment où tous les sites sont dans l'état terminé.

6 Références

[Cha79] E.J.-H. Chang, R. Roberts "An improved algorithm for decentralized extrema finding in circular arrangements of processes" Communication of ACM vol 22 (1979) pp 281-283.

[Dol82] D. Dolev, M. Klawe, M. Rodeh " An O(n log n) unidirectional distributed algorithm for extrema-finding in a circle." Journal of algorithms 3 (1982) pp 245-260.

[Fra82] W.R. Franklin "On an improved algorithm for decentralized extrema finding in circular configurations of processors" Communications of the ACM 25,5 (1982) pp 336-337.

[Kor90] E. Korach, S. Kutten, S. Moran "A modular technique for the design of efficient leader finding algorithms" ACM Transactions on Programming Languages and Systems 12 (1990) pp 84-101.

[Pet82] G.L. Peterson "An O(n log n) unidirectional algorithm for the circular extrema problem." ACM transctions on programming languages and systems 4 (1982) pp 758-762.

[Tar1895] G. Tarry "Le problème des labyrinthes" Nouvelles Annales de Mathématiques 14. (1895)