Suzuki / Kazami : Jeton de permission, une fois le jeton obtenu, on peut y accéder de manière illimité, et on stocke les demandes.

Acquisition :

```
etat_i = demandeur ;
Si (non jetonPresent)
    | nbreq_i[i] = nbreq_i[i] + 1 ;
    | ∀j ≠ i : Envoyer (<REQUEST>) à j ;
    | attendre (jetonpresent_i) ;
etat_i = dedans ;
```

Libération :

```
etat_i := dehors ;
jeton[i] = nbreq_i[i]
∀j de i+1 à n, puis de 1 à i-1 :
    | Si (nbreq_i[j] > jeton[j])
    | alors jetonpresent_i := faux ;
    | Envoyer (<jeton>) à j ;
    | Break ;
```

Reception de Request de j :

```
nbreq_i[j] = nbreq_i[j] + 1
Si : jetonpresent_i et etat_i = dehors et nbreq_i[j] > jeton[j]
    jetonpresent_i = faux
    Envoyer (<Jeton>) à j ;
```

Lors de réception de <Jeton> :

```
jetonpresent_i = vrai ;
```

- $nbreq_i[j]$ = nombres de requêtes faites par j connues de i.
- Seul le possesseur du jeton peut modifier son état
- jeton[i] = nombre d'utilisation de la sc par i.
- jeton[i] < nombre de demande de sc par i.
- 0 message si le site a déjà le jeton. n(n-1 requêtes + jeton) sinon.

Travail et terminaison anneau unidirectionnel :

Lors de reception d'un message de travail :

```
etat_p := actif ;
couleur_p := noir ;
faire travail....
à la fin du travail
etat_p := passif
Si (jetonpresent_p)
    | couleur_p := blanc ;
    | Envoyer (Jeton, 1)
    | jetonpresent_p := faux.
```

Lors de réception d'un message <Jeton, v> :

```
Si (etat_p = actif) alors
    Jetonpresent_p := vrai ;
sinon
    Si (v = n et couleur_p = blanc)
    | terminaison détectée ;
    sinon
        Si (couleur_p = blanc)
            Envoyer (<Jeton, v+1>) ;
        sinon
            couleur_p = blanc ;
            Envoyer (<Jeton, 1>)
        jetonpresent_p := faux
```

$etat_p$: {actif, passif}

couleur : {noir, blanc}

$jetonpresent_p$: {vrai, faux}

- au début un site est désigné par envoyer le jeton vers un suivant
- Lorsqu'un jeton est sur un site actif il ne part avec la valeur 1 que lorsque le site à fini son travail
- Si pendant un tour, que des sites blancs, c'est terminé

- dyssimétrie des processus car p_0 est déterminé.
- Impose structure anneau

Election :

En cas de réveil spontané :

```
participant_i := Vrai
max_i := val_i ;
Envoyer (<Election, val_i>)
```

Lors de réception de message <Election, v> :

```
max_i := max {max_i, v}
Si : val_i = v
    Arret (Succès)
Sinon
    Envoyer <Election, v>
```

Exclusion mutuelle dans un arbre :

```
Procédure acquisition
    Si (non Avoir_jeton_p) alors
        Si (File_vide(Request_p)) alors Envoyer(< REQUEST >) à Racine_p ;
        Ajouter(Request_p, p) ;
        Attendre(Avoir_jeton_p) ;
    En_SC_p := vrai ;

Procédure libération
    En_SC_p := faux ;
    Si (non File_vide(Request_p)) alors
        Racine_p := Défiler(Request_p) ;
        Envoyer(< JETON >) à Racine_p ;
        Avoir_jeton_p := faux ;
        Si (non File_vide(Request_p)) alors
            Envoyer(< REQUEST >) à Racine_p ;

Lors de la réception de < REQUEST > de q
    Si (Avoir_jeton_p) alors
        Si (En_SC_p) alors Ajouter(Request_p, q) ;
        Sinon
            Racine_p := q ;
            Envoyer(< JETON >) à Racine_p ;
            Avoir_jeton_p := faux ;
    Sinon
        Si (File_vide(Request_p)) alors Envoyer(< REQUEST >) à Racine_p ;
        Ajouter(Request_p, q) ;

Lors de la réception de < JETON > de q
    Racine_p := Défiler(Request_p) ;
    Si (Racine_p = p) alors Avoir_jeton_p := vrai ;
    Sinon
        Envoyer(< JETON >) à Racine_p ;
        Si (non File_vide(Request_p)) alors Envoyer(< REQUEST >) à Racine_p ;
```

Naimi et Trehel

Procédure demander sc :

```
demandeur_p := vrai
Si (pere_i ≠ nil) alors
    Envoyer (<Request>) à père_p ;
    pere_p := nil
    Attendre (Avoir Jeton_p)
```

Libération, réception de Request de q (q demandeur) :

```
(Demandeur_p) alors
    (Suivant_p = nil) alors suivant_p := q
Si (pere_p ≠ nil) alors
    Envoyer (<Request>) à père_p
Sinon si (AvoirJeton_p) alors
    AvoirJeton_p := faux
    Envoyer (<Token>) à q
père_p := q
```

Procédure recevoir <Token> de q :

Initialisation : Choisir sommet S quelconques

```
AvoirJeton_s := vrai ;
Pour tout X ≠ S, AvoirJeton_x := faux ;
Pour tout sommet Y, pere_y := S ; suivant_y := Nil ; Demandeur_y := faux ;
```

Chang-Roberts : Election anneau

En cas de réveil spontané :
participant$_i$:= vrai
Envoyer (< Election, val$_i$ >);

Lors de la réception d'un message <Election, v>
Si (v > val$_i$) alors
participant$_i$:= vrai
Envoyer (< Election >);
Si (v < val$_i$) et (non participant$_i$) alors
participant$_i$:= vrai;
Envoyer (< Election, val$_i$ >)
Si (v = val$_i$) alors Envoyer (< ELU, val$_i$ >)

Lors de la réception d'un message <Elu, v> :
Le gagnant est le site v;
participant$_i$:= Faux
Si (val$_i$ ≠ v) alors Envoyer (< Elu, v >)

O(n log n)

Hirschberg et Sinclair :

Phase i :
Tant que (Etat == actif)
Envoyer < val$_i$, 2^i >;
i := i+1

Reception Message <v, TTL> :
Si (Etat == actif) et (TTL != 1) alors
Faire transiter le message
Sinon
Si v > val$_i$ alors état = passif
Sinon si v = val$_i$ alors Elu.

au alors

Reveil par signal etat = actif
i := 0
Tant que (etat = actif)
Envoyer (election, val$_i$, 2^i)
Attendre (reception message)

Reception du message (election, V, D) sur j
Si etat = actif
Si v > val$_j$:
Alors état = perdant;
Si (v = val$_j$)
etat = gagnant
Si etat = perdant
Si D-1 > 0
Envoyer (election, V, D-1).

Ecrire un algorithme réparti qui :

— "Réveille" tous les sommets à partir d'au moins un *initiateur*.

— Calcule la plus petite étiquette de l'arbre (le gagnant étant le sommet qui possède cette étiquette). Pour cela :

— Faire un parcours des feuilles vers l'intérieur de l'arbre en calculant la plus petite étiquette au fur et à mesure.

— Propager le résultat de l'élection de voisin en voisin. Si un sommet reçoit son étiquette, il se déclare *vainqueur*, sinon *perdant*.

Pour cela vous pourrez utiliser les variables suivantes :
Un site p a les variables locales suivantes :
$reveille_p$ booléen initialisé à $Faux$,
$voisins\_reveilles_p$ entier initialisé à 0,
rec_p tableau de $|voisins_p|$ booléens initialisés à $Faux$,
val_p la valeur pour l'élection, v_p, initialisée à la valeur val_p,
$etat_p \in \{neutre, gagnant, perdant\}$, initialisé à $neutre$.

4n - 4 messages

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

```
Si (p est initiateur) alors
    reveille_p := Vrai;
    Pour tout (q ∈ voisins_p) faire
        Envoyer(< REVEIL >) à q;
Lors de la réception d'un message pour la première fois
 (*** PHASE DE REVEIL : ***)

    Tant que (voisins_reveilles_p < |voisins_p|) faire
        Attendre Recevoir message(< REVEIL >);
        voisins_reveilles_p := voisins_reveilles_p + 1;
        Si (non reveille_p) alors
            reveille_p := Vrai;
            Pour tout (q ∈ voisins_p) faire
                Envoyer(< REVEIL >) à q;
 (*** PHASE DE CALCUL : ***)

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

    Attendre Recevoir(< TOKEN, r >) de q_0;
    v_p := min{v_p, r};
    Si (v_p = val_p) alors etat_p := gagnant;
    Sinon etat_p := perdant;
    Pour tout (q ∈ voisins_p, q ≠ q_0) faire
        Envoyer(< TOKEN, v_p >) à q;
```

Carvalho Roucairol :

```
Lors d'un appel à acquérir
    etat_i = demandeur;
    last_i = h_i + 1;
    ∀j ∈ R_i : envoyer requête(last_i, i) à j;
    attendre (R_i = ∅);
    etat_i = dedans;

Lors d'un appel à liberer :
    etat_i = dehors;
    ∀j ∈ differe_i : envoyer permission(i, j) à j;
    R_i = differe_i;
    differe_i = ∅;

Lors de la reception de requete(k, j)
    h_i = max(h_i, k);
    priorité_i = (etat_i = dedans) ou ((etat_i = demandeur) et (last_i, i) < (k, j));
    Si priorité alors diffr_i = diffr_i ⋃ {j};
    Sinon;
        envoyer permission(i, j) à j;
        R_i = R_i ⋃ {j};
        Si etat_i = demandeur;
        Alors envoyer requete(last_i, i) à j;
        fsi
    fsi

Lors de la reception de permission(i, j);
    R_i = R_i − {j};
```

Agrawala

```
Procédure acquisition
    demandeur_i := vrai;
    horloge_i := horloge_i + 1;  heure_demande_i := horloge_i;
    rep_attendues_i := n − 1;
    Pour tout (x_i ∈ V − {i}) faire
        Envoyer(< REQUEST, heure_demande_i >) à x_i;
    Attendre(rep_attendues_i = 0);
Procédure libération
    demandeur_i := faux;
    Pour tout (x_i ∈ V − {i}) faire
        Si (differe_i[x_i]) alors
            Envoyer(< REPONSE >) à x_i;
            differe_i[x_i] := faux;
Lors de la réception de < REQUEST, h > de j
 horloge_i := max{horloge_i, h};
    Si (non demandeur_i ou (heure_demande_i, i) > (h, j)) alors
     Envoyer(< REPONSE >) à j;
 Sinon differe_i[j] := vrai;
Lors de la réception de < REPONSE > de j
    rep_attendues_i := rep_attendues_i − 1;
```

Algorithmes :

- Sûreté : À chaque instants, au plus m sites utilisent la ressource.
- Vivacité : Chaque site demandant un accès à la ressource l'obtiendra au bout d'un temps fini. (Absence de famine).

Algorithme de Lamport : Utilisation d'estampilles par avoir un ordre total. 3 types de messages : Les requêtes, la libération et l'acquitement.
Acquitement = accusé de réception d'une requête.

Demande d'accès à la ressource i :

Diffuser (Requête, h, i)
T[i] ← (Requête, h, i)

Lors d'une réception d'un message du type (type, k, j)

h ← max(h, k) + 1,
T[j] ← (type, k, j)
Si (T[i].type ≠ requête)
| si (type == requête) envoyer (acquitement, h, i) à j

Sinon si (∀j ≠ i, (T[i].date, i) < (T[j].date, j))
| acceder à la sc

Libération :

Diffuser (Libération, h, i)
T[i] ← (Libération, h, i)