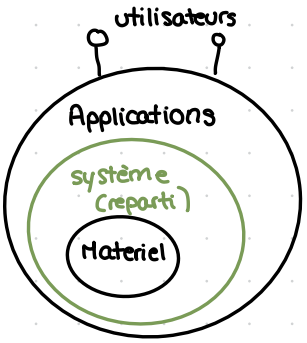
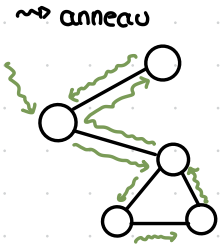
Cours 3 21/01/26



1) Communication

diffusion
récolte (gathering)
construction de structures



2) Contrôle

- Election de leader
- Exclusion mutuelle
- detection de terminaison
- snapshots (instantanés)

3) Temps (séquencement)

- horloges (logiques)

4) Tolérance aux défaillances

- Autostabilisation

Complexité:
- en messages
  - nbr de messages émis / bits émis
- en temps

→ Algorithme diffusion (nouvelle)

type canal = entier
voisins : ensemble de canal;
déjà_vu: booléen initialisé à faux

un processus sur réception de Nouvelle:
si déjà_vu = faux alors
  pour tout c dans voisins (c, Nouvelle);

Diffusion de Nouvelle sur un arbre préalablement construit (fils) avec comme racine le processus recevant Nouvelle:
fils: ensemble de canal;
racine booléen initialisé à faux sauf pour le processus recevant Nouvelle,
le processus racine sur réception de nouvelle →
  si fils ≠ ∅ alors
    pour tout c dans fils envoyer (c, Nouvelle)
un processus sur réception de Nouvelle →
  si fils ≠ ∅ alors
  pour tout c dans fils envoyer (c, nouvelle)

* missing: cours 3

Cours 4 28/01/26

Diffusion sur un arbre avec détéction de terminaison par la racine

constantes: fils = ensemble de canal;
père = canal

variables: racine: booléen initialisé à faux sauf pour un processus;
attendus: ensemble de canal

la racine sur réception de nouvelle →
si fils = ∅ alors <terminer> sinon
attendus := fils;
pour tout c dans fils envoyer(c, Nouvelle);

attendus = {1, 32}
1) racine
1 32
18 fin 22 attendus = {18, 22}

fin = aquittement
un processus sur reception de Nouvelle →
si fils = ∅ alors envoyer (père fin)
sinon attendus := fils; pour tout c dans attendu envoyer(c, fin(Nouvelle));

un processus sur réception de fin (Nouvelle) sur le canal c →
attendus := attendus - {c};
si attendus = ∅ alors
si racine alors <terminer>
sinon envoyer (père, fin (Nouvelle)

nouvelle : 1 par arrête
aquittement : 1 par arrête
) n ⇒ 2n
ne peut pas être executée avec moins de n message (sinon ∃ un processus qui n'a pas reçu de message)

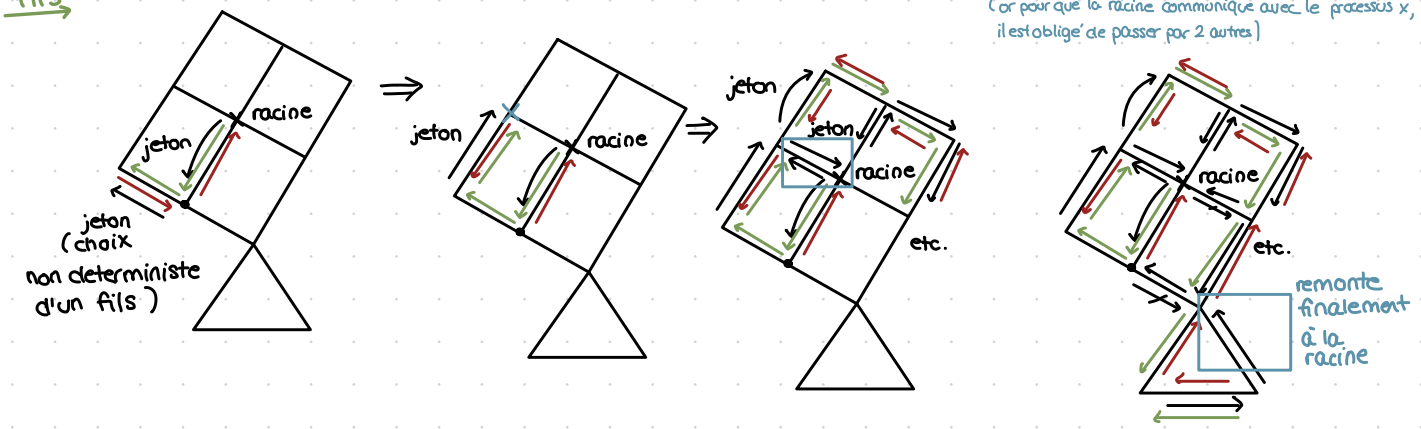
II. Comment construire un arbre recouvrant d'un graphe connexe quelconque?

1) Construction par exploration séquentielle: (Audio 1)

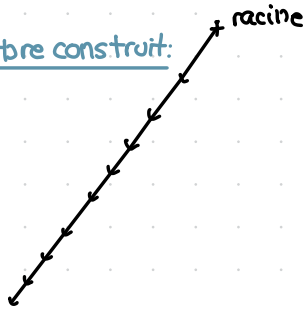
Le jeton visite successivement tous les processus.

"aquittement pour signaler qu'il ne veut pas être fils de la racine"
↳ séquentiellement, il est plus probable de tomber sur un processus sans père)

père → fils



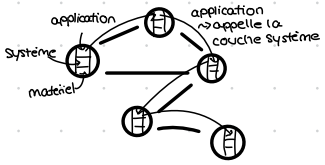
arbre construit:



code associé:

constantes et variables

voisins = ensemble de canal;
initiateur, participant : booléen initialisé à faux;
encore, fils : ensemble de canal;
père : canal;



un processus sur réception de Unit →

```
initiateur := vrai;
participant := vrai;
si voisins = ∅ alors terminer
sinon pourtout c dans voisins envoyer (c, Vu); □
    choisir c dans voisins
    envoyer (c, exploration);
    fils := {c};
    encore := encore - {fils};
```

un processus sur réception de Vu sur reception de Vu sur le canal c →

```
encore = encore - {fils};
```

un processus sur réception de exploration sur le canal c →

```
si non participant alors | participant = vrai;
                         | père = c;
                         | pour tout c dans voisins envoyer(c, Vu);
                         | encore := encore - {père};
                         si encore ≠ 0 alors
                             choisir c' dans encore;
                             envoyer (c', exploration);
                             fils := {c'};
                             encore := encore - fils;
                         sinon envoyer (père, retour);
sinon envoyer (c, annule);
```

un processus sur réception de retour ou d'annule sur le canal c →

```
si annule alors fils := fils - {c}
si encore ≠ ∅ alors
        choisir c dans encore
        envoyer (c, exploration);
        fils := fils ∪ {c};
sinon si initiateur alors <terminer>
      sinon envoyer(père, retour);
```

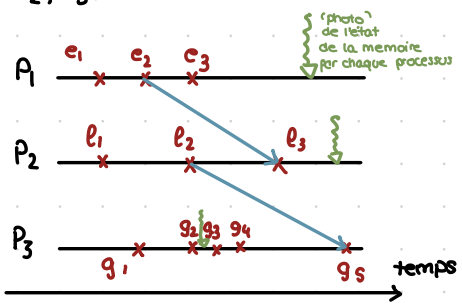
Cours 5 04/02/26

Horloges logiques

horloge: application d'un ensemble d'événements dans un ensemble (partiellement) ordonné, en relation avec une relation de causalité

processus séquentiel:

$a_1, a_2, a_3, \ldots$



événements : entiers | vecteurs d'entiers

$a \longrightarrow \theta(a)$

Une horloge est compatible avec une relation de causalité si $a \rightarrow b \Rightarrow \theta(a) < \theta(b)$

L'horloge est caractéristique de la causalité si $a \rightarrow b \Leftrightarrow \theta(a) < \theta(b)$

Deux événements a et b sont indépendants ssi on n'a ni $a \rightarrow b$, ni $b \rightarrow a$

Horloge de Fidge - Mattern

Histoire de a, $H(a)$ (k premiers événements d'un processus)

$H(a) = \{b \mid b \rightarrow a\} \cup \{a\}$

↑ passé causal de a

Projections $H_1(a), H_2(a), \ldots$,

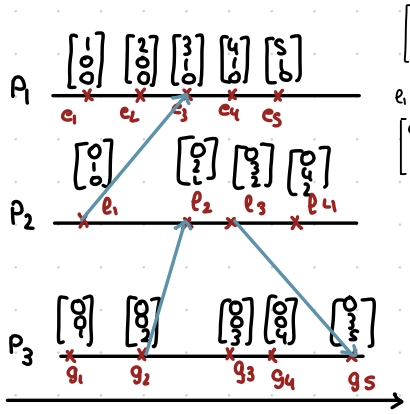
$$\theta(a) = \begin{bmatrix} \text{card } H_1(a) \\ \text{card } H_2(a) \\ | \end{bmatrix}$$

(* pour comparer 2 vecteurs :

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \leq \begin{bmatrix} t \\ u \\ v \end{bmatrix} \Leftrightarrow \begin{bmatrix} x \leq t \\ y \leq u \\ z \leq v \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} < \begin{bmatrix} t \\ u \\ v \end{bmatrix} \Leftrightarrow \begin{bmatrix} x \leq t \\ y \leq u \\ z \leq v \end{bmatrix} \text{ et } \neq \Big)$$

diagramme de causalité:



e_5 et g_5 : $\begin{bmatrix} 5 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 3 \\ 5 \end{bmatrix}$ incomparables

l_1 et g_5 : $\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} < \begin{bmatrix} 0 \\ 3 \\ 5 \end{bmatrix}$

preuve: $\Rightarrow$ transitivité

$$a \rightarrow b \Leftrightarrow H(a) \subsetneq H(b) \text{ et } a \neq b$$

$$\Leftarrow \quad H_1(a) \subseteq H_1(b)$$

$$\Leftrightarrow \quad H_2(a) \subseteq H_2(b) \text{ et } a \neq b$$

$$\Leftrightarrow \quad \text{Card}(H_1(a)) \leq \text{Card}(H_1(b))$$

$$\text{Card}(H_2(a)) \leq \text{Card}(H_2(b)) \text{ et } a \neq b \quad \Leftrightarrow \begin{bmatrix} H_1(a) \\ H_2(a) \\ | \end{bmatrix} \not\leq \begin{bmatrix} H_1(b) \\ H_2(b) \\ | \end{bmatrix}$$

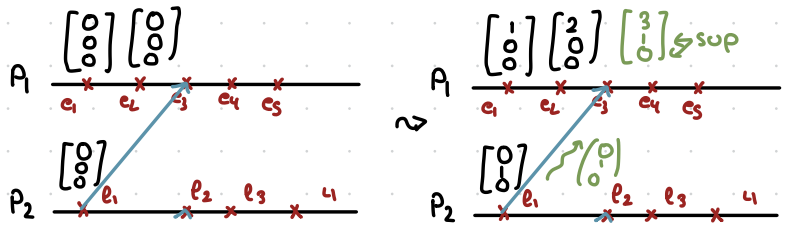
Calcul de H_i par le processus P_i } algo

Initialement $H = \begin{bmatrix} 0 \\ 0 \\ | \\ 0 \end{bmatrix}$

(réception locante)

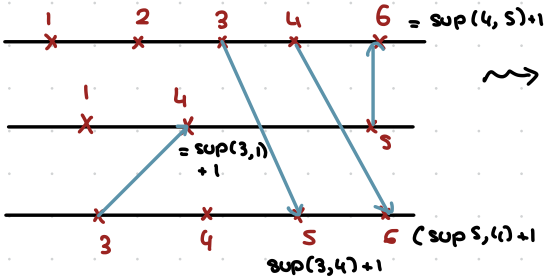
1) Lors d'un événement interne, incrémenter la $i^{ème}$ composante de H
2) Lors d'une émission d'un message m, incrémenter la $i^{ème}$ composante de H et attacher H à m (piggybacking)
3) Lors de la réception d'un message m, incrémenter la $i^{ème}$ composante de H et prendre le sup des autres composantes.

exemple:



Horloge de Lamport:

(ajouter 1 à chaque chemin)



valeur de l'horloge de Lamport de l'evenement a

$\rightsquigarrow$ si $a \rightarrow b$ alors $L(a) \not\leq L(b)$

si on considère $(L(a), i)$

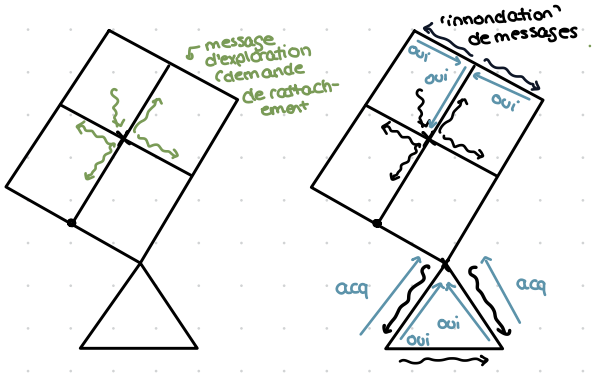
$(L(a), i) \not\leq (L(b), i) \Leftrightarrow L(a) < L(b)$

ou $L(a) = L(b)$ et $i < j$

$\rightsquigarrow$ permet d'avoir un ordre total sur les évenements

Suite cours 4 (Construction d'arbre)

Construction d'arbre par exploration parallèle



code:

variables / constantes:

voisins : ensemble de canal

participant : booléen initialisé à faux

attendus, fils : ensemble de canal

père : canal

racine : booléen initialisé à faux

un processus sur réception de Init →

participant := vrai ; racine := vrai ; fils = ∅ ;

attendus := voisins ;

on supp graphe non-dégénéré

pour tout c dans voisins envoyer (c, explorer) ;

un processus sur réception de Explorer sur le canal c →

si non participant alors | participant := vrai ;
| père := c ;
| fils := ∅ ;
| attendus := voisins - {c} ;
| si attendus ≠ ∅ alors pour tout c' dans attendus
| envoyer (c', explorer)
| sinon envoyer (père, oui)

sinon envoyer (c, Non) ;

un processus sur reception de oui sur le canal c →

fils := fils ∪ {c} ;

attendus := attendus - {c} ;

si attendus = ∅ alors

si racine alors <terminer>

sinon envoyer (père, oui) ;

un processus sur réception de Non sur le canal c →

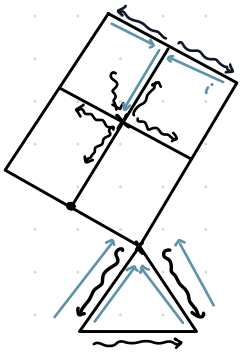
attendus := attendus - {c} ;

si attendus = ∅ alors

si racine alors <terminer>

sinon envoyer (père, oui) ;

Cours 6 11/02/26



attendus: acquittements de vagues
encore: acquittements de terminaux

Bellman-Ford

Algorithme de construction d'arbre recouvrant par vagues synchronisées et détection de terminaison

variables / constantes:
voisins ;
encore, attendus, fils : ensemble de canal;
père : canal ;
niveau : entier;
racine : booléen initialisé à faux ;

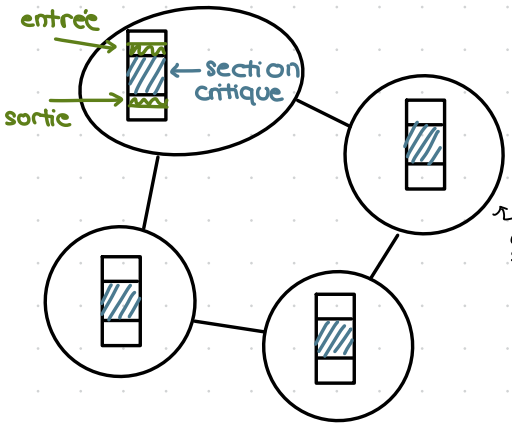
Initiallement, l'environnement contacte un processus.

Un processus sur communication avec l'environnement →
participant := vrai; racine := vrai;
niveau := 0; encore := voisins; ← pour avoir la terminaison
si encore ≠ 0, alors pour tout c dans voisins envoyer(c, aller(1)) ← vague à distance 1 de la racine
attendus := voisins;

un processus sur réception de aller(K) sur le canal c →
si non participant alors | participant = vrai; père := c; niveau := K; fils = ∅;
encore := voisins - {père};
si encore = ∅ alors envoyer(père, retour(fin(K))
sinon envoyer(père, retour(again(K)));
sinon // si participant
si père = c alors | père := voisins - {père};
pour tout c dans encore
envoyer(c, aller(K));
attendus := encore
sinon envoyer(c, retour(non(K)));

un processus sur réception (retour(reponse(K)) sur le canal c →
attendus := attendus - {c};
si(réponse = non) ou (réponse = fin) alors encore := encore - {c};
si(réponse = again) ou (réponse = fin) alors fils := fils ∪ {c};
si encore ≠ 0 et attendus = ∅ alors
si(¬ racine) alors envoyer(père, retour(again(K))
sinon | pour tout c dans encore envoyer(aller(K+1));
attendus := encore;
si encore = 0 alors si(non racine) alors envoyer(père, retour(fin(K));
sinon < terminé >

Exclusion mutuelle des sections critiques.



un processus est dans sa section critique si son compteur de programme pointe vers une instruction de la section critique.

↳ on veut que l'accès à la section critique soit séquentiel

↳ à chaque instant, au plus un processus exécute une instruction dans sa section critique (⇔)

↳ causalement indépendants

Spécification précise d'un problème :

≡ propriété d'exécution

≡ conditions sur les exécutions

Safety (sureté) = satisfiabilité à chaque partie d'exécution (⇔) ↳ ne pas construire un arbre en cycle par exemple

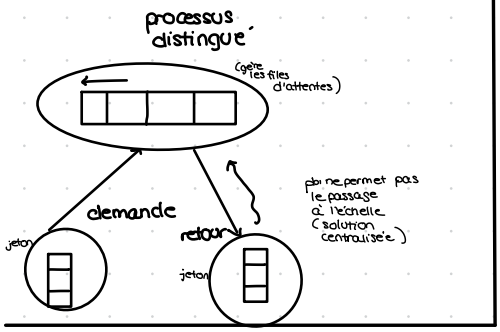
Liveness (vivacité) = satisfaite dans le futur

↳ un processus peut rentrer dans sa section critique au bout d'un temps fini

↳ terminaison par exemple

(ex (pour élection de leader)
S : à aucun moment on a 2 ou plus leaders
E : terminaison

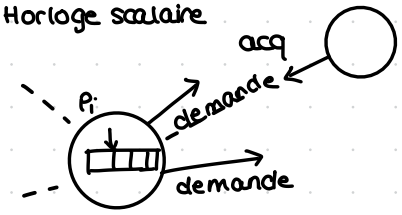
Solution centralisée :



Horloge de Lamport avec l'ordre total :

(∼, i) ↓ ↳ num du processus
↓ valeur de l'horloge de Lamport

Horloge scalaire



canaux FIFO (on conserve l'ordre des messages)

Avant d'entrer en section P_i, un processus attend que l'estampille (↓ le couple) soit la plus petite de toutes les estampilles des messages qu'il a reçu impliquant que la demande est causalement la plus ancienne et unique

---

type_de_message : demande, sortie, acquittement

message : 3 champs : type
{ horloge
{ numéro de processus

file_d_attente : tableau [1, ..., n] de message ; (initialisé à file[i] = (sortie, 0, i);

un processus P_i pour demander à rentrer en S.C ⟶

```
gérer l'horloge;
diffuser(demande, horloge, i);
attendre file[i] contienne le plus petit couple
(estampille, numéro) de toute la file d'attente;
dedans := vrai;
< section critique >
```

un processus qui reçoit un message de demande de p_i →

tant que dedans attendre

gerer l'horloge;

envoyer(acquittement, horloge, i);

un processus A_i qui sort de section critique →

gérer l'horloge; (incrémenter)

diffuser(sortie, horloge, i);

$\forall j$,

à la réception de {acquittement, sortie, j} →

si la cellule j ne contient pas une demande, mettre à joure file(j);

Cours 7 18/02/26

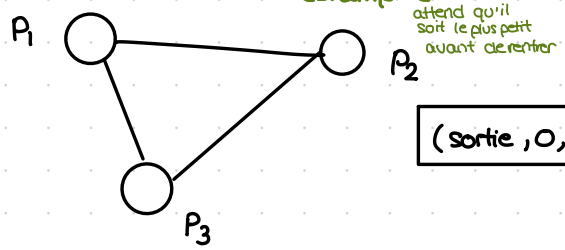
Algorithme d'exclusion mutuelle de Lamport (exemple de déroulement de l'algo)

Tableau demande : P_1 P_2 P_3

(sortie, 0, 1)	(sortie, 0, 2)	(sortie, 0, 3)

estampille

attend qu'il soit le plus petit avant de rentrer



(sortie, 0, 1)	(sortie, 1, 2)	(sortie, 0, 3)

(sortie, 0, 1)	(sortie, 0, 2)	(sortie, 0, 3)

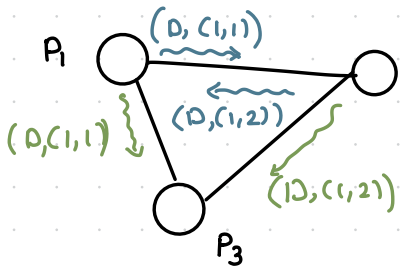
P_1 demande à entrer en S.C.

P_2 demande à entrer en S.C.

Durant toute l'execution, P_3 ne demande pas à entrer en S.C.

⇒

(demande, 1, 1)	(demande, 1, 2)	(sortie, 0, 3)



Δ! canaux FIFO (l'ordre des demandes est conservé)

(demande, 1, 1)	(demande, 1, 2)	(sortie, 0, 3)

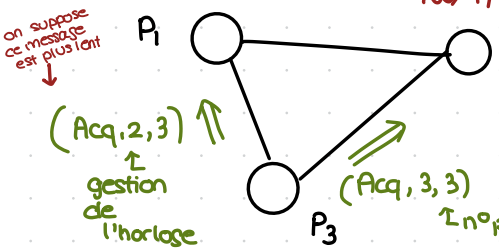
(demande, 1, 1)	(sortie, 0, 2)	(sortie, 0, 3)

⇒

(demande, 1, 1)	(demande, 1, 2)	(acq, 2, 3)

P_1 en SC

⇝ P_1 entre en section critique



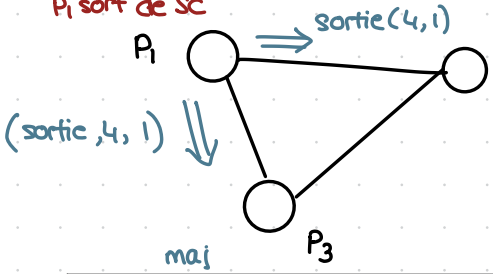
P_2 est tjs bloqué

(demande, 1, 1)	(demande, 1, 2)	(acq, 3, 3)

(demande, 1, 1)	(sortie, 0, 2)	(sortie, 0, 3)

(demande, 1, 1)	(demande, 1, 2)	(acq, 2, 3)

P_1 sort de SC



(sortie, 4, 1)	(demande, 1, 2)	(acq, 3, 3)

⇝ P_2 entre en SC

maj

sortie(4, 1)	(sortie, 0, 2)	(sortie, 0, 3)

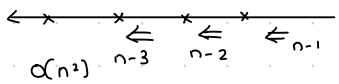
Complexité:

en messages

nb de messages émis pour 1 entrée en S.C :

→ (n-1) (P1 diffuse son message)
+ (n-1) (diffuse son message de sortie) ⇒ 3(n-1)
+ (n-1) (acquittements)

(nb d'arrêtes)
(n-1 si graphe complet)



Algorithme de Ricard et Agrawala:

Privilège: avoir la plus petite estampille parmi les demandes

↳ exclusion mutuelle est réalisée ⇝ seul un processus avec une plus petite peut entrer en S.C. (algo + FIFO)
la plus petite demande est unique (Lamport'élargi)

Faimess: Le processus avec la plus petite estampille finit par entrer en S.C.
(pas de perte de message) (pas de blocage)
- processus "précédé" par les autres processus
  - i) K < n-1 (algorithme)
  - ii) au bout d'un certain temps, K = K-1

Privilège implicite (condition)
R.A. privilège explicite ⇝ jeton

Le 'jeton' est un tableau qui contient le nb d'entrées en S.C. de chaque processus

jeton

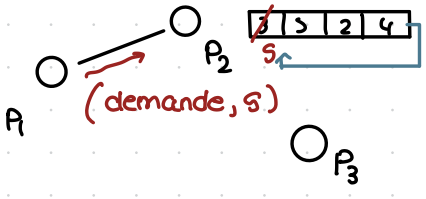
2	0	5	8

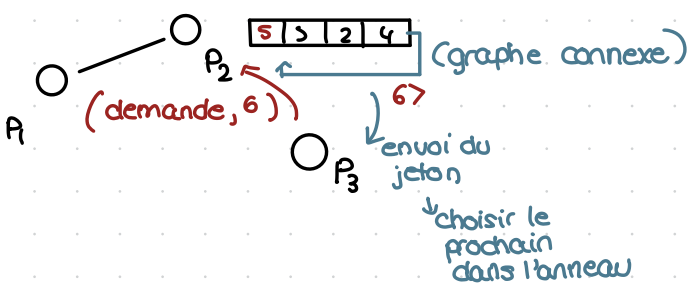
(horloge interne de chaque processus)

- Lorsqu'un processus qui "veut" entrer en S.C. libère le jeton à la sortie de S.C., il incrémente son compteur (dans le jeton)
- Au départ, le jeton est placé dans un processus particulier, et toutes ses cellules contiennent 0.

Si la possession du jeton garantit l'entrée en S.C et si une entrée en S.C. est impossible sans possession du jeton, l'unicité du jeton garantit l'exclusion mutuelle des S.C.

- Un processus qui veut entrer en S.C. diffuse un message de demande, avec le compteur de ses S.C. passées.





---

algo: ← horloge ↓ n processus

```
message = (entier, émetteur) ;
jeton = tableau [site] de entier;
procédure attendre_jeton (v:jeton)
        { attend l'arrivée d'un message de type jeton et range sa valeur dans la variable v }
tampon
```

constantes, variables

{ pour p_i }

const : numéro initialisé à i ;

var : tampon ; demande : jeton ;

estampille : horloge initialisé à 0 ;

jeton_présent : booléen initialisé à (mon_numéro = 1) ;

dedans : booléen initialisé à faux ;

requête : message ;

lors de demande d'entrée en SC →

```
si jeton_présent = faux alors
        début
        | estampille := estampille + 1 ;
        | requête.date := estampille ;
        | requête.émetteur := mon_numéro ;
        | diffuser(requête) ;
        | attendre_jeton(tampon) ;
        fin
    jeton présent := vrai ;
    dedans := vrai ;
```

lors de sortie en S.C →

```
var j : entre site ;
tampon(mon_numéro) := estampille ;
dedans := faux ;
pour j := mon_numéro+1 jusqu'à n, 1 jusqu'à mon_numéro-1
faire :
    si demande(j) > tampon(j) et jeton_présent
    alors | jeton_présent := faux ;
          | envoyer(tampon, j) ;
```

Lors de l'arrivée d'une requête →

```
var h : numéro ;
k := requête_émetteur ;
demande[k] := max( demande[h], requête.date) ;
                  (cas où il reçoit une demande "périmée")
si jeton_présent et dedans = faux alors
    { }
```

→ complexité : en message : $(n-1)+1 = n$

exclusion mutuelle : exclusion du jeton dont la posession est une condition nécessaire et suffisante d'entrer en section critique

Fairness : exploration circulaire

Election sur un anneau

but : désigner un processus particulier, parmis des processus numérotés

spécialisation : Fairness : l'algorithme se termine explicitement

Safety : jamais 2 leaders et 1 seul à la terminaison.

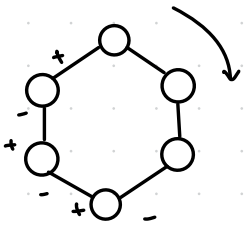
→ graphe en forme d'anneau : - unidirectionnel (successeur)
- bidirectionnel (successeur + prédecesseur)
- orienté ssi étant donnés 2 voisins p_i et p_j, mon_successeur(p_i) = p_j ⟺ mon_prédecesseur(p_j) = p_i

Algorithme de Champ et Roberts

Cours 8 11/03/2026

Élection de leader

Réseau : anneau



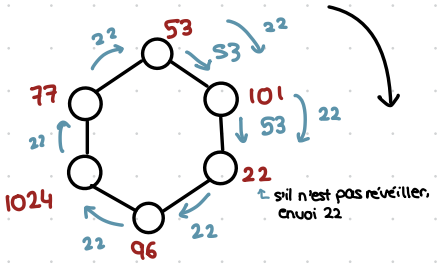
Protocole :
- Terminaison (implicite ou explicite)
- Jamais plusieurs leaders et un seul à la terminaison.

Algorithme de Chang et Roberts (extinction sélective)

L'élu est le processus avec le plus petit identificateur (entiers)

cas asynchrone. anneau orienté

Lors d'un reveil un processus va "tenter" de faire parcourir à son numéro le tour de l'anneau
on fait en sorte que le plus petit numéro peut faire le tour



Pseudo-code :

```
constante : mon_numero
variable : participant initialisé à faux
leader : booléen initialisé à faux
Lors de reveil (par la couche supérieure) →
si non participant alors
    envoyer(+, mon_numero);
    participant := vrai;

Lors de réception de (+, val) sur le lien ←
Si non participant alors
    participant := vrai
    si val < mon_numéro alors envoyer(+, val);
    sinon envoyer (+, mon numero)

sinon si val < mon_numéro alors
          envoyer(+, val);
      si  val = mon_numéro alors
          leader := vrai
          <STOP>
    // sinon on ne fait rien
```

Preuves :

1) Terminaison :

- le processus avec le plus petit numéro se réveille (à un certain moment)

On considère la plus petite distance avec un processus qui se réveille et le processus avec le plus petit numéro
↳ induction sur la distance

- le plus petit numéro fait (au bout d'un certain temps) le tour de l'anneau

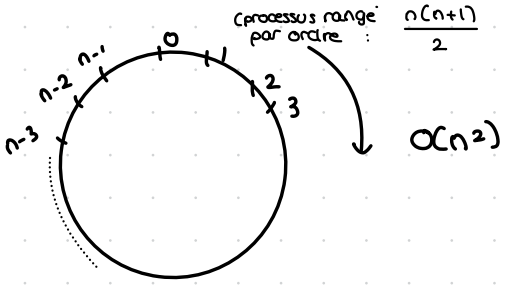
(induction sur la distance parcourue par le plus petit numéro)

2) Complexité

en messages en fonction du nombre de processus : n^2

borne supérieure : n x n

une exécution particulière qui atteint cette borne :



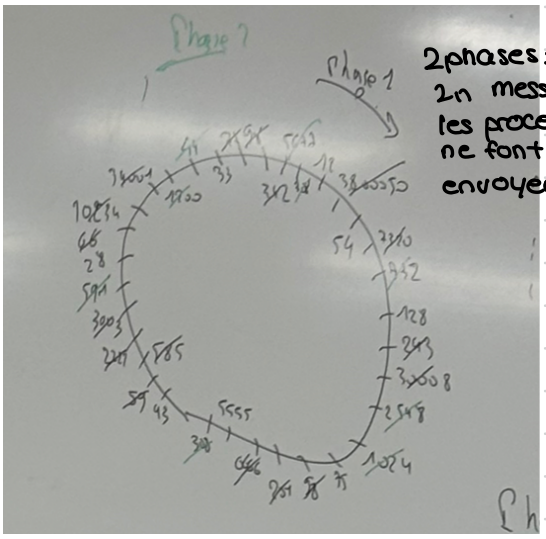
Algorithme de Peterson

anneau bidirectionnel (mais orienté)

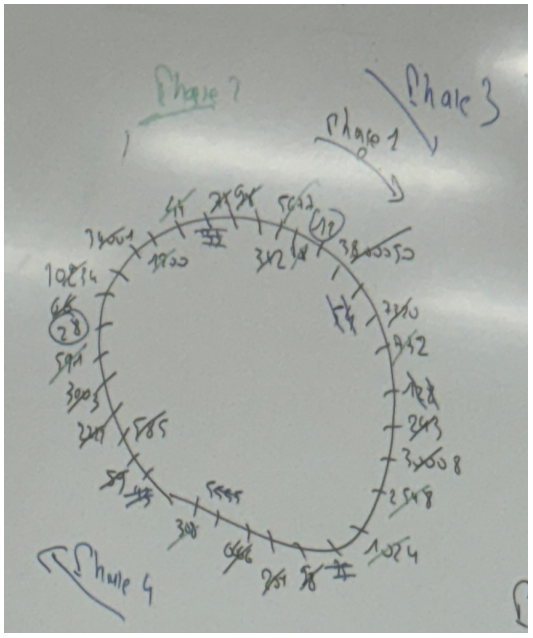
phase: execution où le processus envoi+ reçoit (dans un sens ou l'autre)

Chaque phase fait des victimes et laisse des survivants

(TODO)



2phases :
2n messages
les processus (morts) ne font que envoyer les messages



5phases, 5n messages

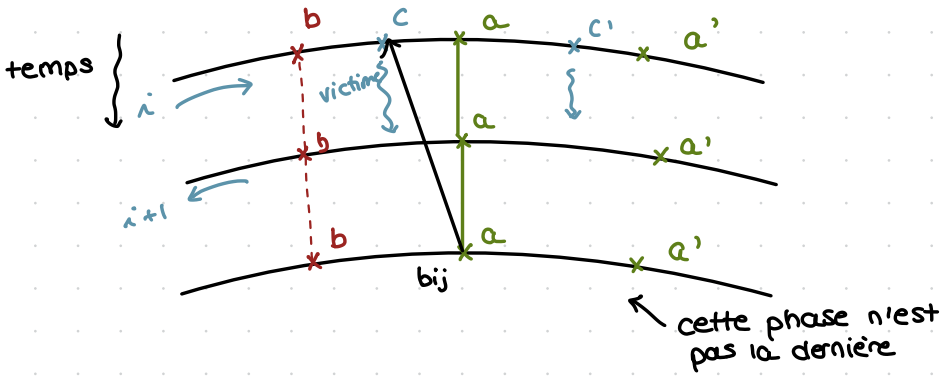
Idée : 1. Une exécution procède par une succession de phases, alternant les directions + et -

2. Une phase est un segment d'exécution dans lequel un processus émet un message et reçoit un message ou reçoit un message et envoyer un message. (rq: chaque processus a son propre rythme de phase)
3. Chaque phase fait des victimes et laisse des survivants. Une victime est un processus qui reçoit une plus petite valeur que son numéro Par la suite une victime ne fait que relayer les messages
4. Lorsqu'il termine une phase, un processus survivant entame la phase suivante en envoyant son numéro dans la direction opposée à celle de la phase précédente
5. Un processus qui reçoit son numéro devient leader

Eléments de preuve :

1. Le nombre de phases est fini
2. Au bout d'un certain temps, le plus petit numéro fait le tour de l'anneau.

Toutes les deux phases, le nombre de processus vivants au début des deux phases a été divisé par au moins 2 à la fin des 2 phases. (nb de phases $O(\ln(n))$



a est le numéro d'un processus
Durant la phase $i+1$, a a reçu une valeur plus grande
Durant la phase i, a a reçu une valeur b et puisque a est vivant, $b > a$

b est le plus petit numéro vivant à la fin de la phase $i+1$ dans le sens − par rapport à a

→ il existe une injection ($c = c'$) entre les processus survivants à la fin d'une phase et les victimes faites dans les 2 phases précédentes
⇒ le nb de processus est divisé par 2 après

Cours 9 18/03/2026

Detection de terminaison

Le système utilise des messages de contrôle.
Il doit annoncer que l'application est terminée, si c'est le cas
↳ le processus de l'application
sont passifs et il n'y a pas de travail en transit.

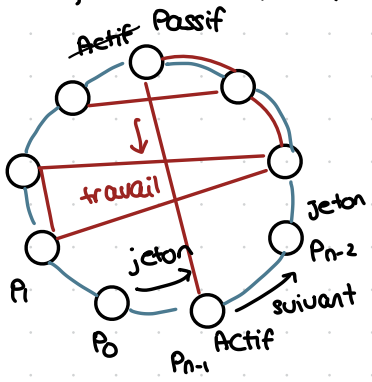
Problème: On a une application distribuée qui échange des messages (messages de travail)
Quand un processus reçoit un message de travail, il devient "actif"
S'il ne reçoit pas d'autres messages de travail, un processus actif devient passif dans le futur.

Sureté: Le système n'annonce pas de fausse terminaison
Vivacité/Equité: le système doit l'annoncer au bout d'un nombre fini d'opérations.

Hyp: anneau orienté unidirectionnel, communications instantanées

Preuve par invariant:
↳ prédicat : - vrai dans la configuration initiale
- quand il est vrai dans une certaine configuration, cela implique que le problème est résolu

Règles:
R_1: Un processus a une couleur, blanc ou noir, initialement blanc
R_2: Le jeton a une couleur, blanc ou noir, intitialement blanc.
R_3: Un processus qui reçoit un message de travail devient noir
R_4: Seuls les processus passifs transmettent le jeton
R_5: Un processus blanc transmet le jeton
Un processus noir transmet le jeton noir, et redevient blanc
R_6: Si le jeton revient blanc à P_0, lui même blanc et passif, P_0 <annonce> la terminaison. (...)



Pseudo-code de l'algo de Djikstra :

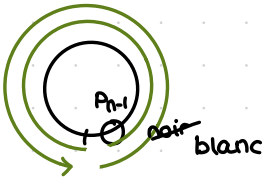
Application

lors d'un envoi d'un message par p de travail reçu par q → état := actif (transmission instantanée)
spontanément → état := passif;

lors de l'envoi d'un message de travail par p reçu par q →
couleur$_p$:= noir;
état$_q$:= actif;
quand état = actif → état := passif
lors de l'initiation d'une détection de terminaison par p_0 →
envoyer (jeton, blanc) à P_{n-1} ;
quand p reçoit le jeton (jeton, c) et est passif →
si p = p_0 alors
si (c = blanc) et (couleur p_0 = blanc) alors
<annoncer>
Sinon envoyer jeton blanc à P_{n-1}
sinon si (couleur$_p$ = blanc) // et est passif alors
envoyer (jeton, c) à suivant sur l'anneau
sinon // la couleur de p est noir
envoyer (jeton, noir) à suivant
couleur$_p$:= blanc

Complexité:

nombre maximum de messages de contrôle après que l'application est terminée (latence).



2 tours de l'anneau (-1)

Algorithme de Safia (transmission des messages asynchrones)

Utilisation de Compteurs de message

(règles de Djikstra+:)

R1: Chaque processus a un compteur de la différence entre nombre de messages de travail émis et nombre de messages de travail reçu

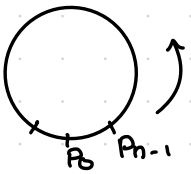
R2: le jeton relève les compteurs.

Cours 10 25/03/2026

Detection de terminaison (Safia)

{ pour le processus p }

var state = (actif, passif);
color : (blanc, noir);
me : entier init à 0;



quand state = actif ⟶
envoyer (message);
me := me + 1;

quand un message mes arrive ⟶
recieve (mes);
me := me - 1;
color := noir;

quand état = actif ⟶
etat := passif;

pour lancer une détection de terminaison ⟶
envoyer(tok, blanc, 0) à P_{n-1} ;

pour un processus pour traiter le token <tok, c, q>
quand état = passif ⟶
si p = p0 alors si <c = blanc> et colorp = blanc et mep + q = 0 alors <Annoncer>
sinon envoyer(tok, blanc, 0) à P_{n-1}
sinon si color = blanc alors envoyer(tok, c, q + mep) à suivant
sinon envoyer(tak, noir, q + mcq) à suivant;
color := blanc;

(le jeton fait au plus 2 tours après la terminaison)

Gerard Tel - Distributed algorithm - pu par invariant

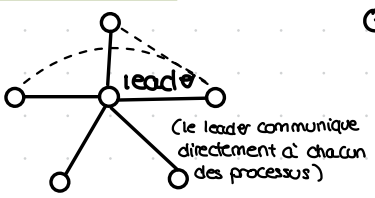
Seconde version des compteurs (Mattern)

(au plus 1 tour après la terminaison)

$$\begin{bmatrix} mc_1 \\ mc_2 \\ \vdots \\ mc_{n-1} \end{bmatrix}$$

Algorithme du crédit (Mattern)

Calcul diffusant :



Quand le leader reçoit le signal de l'application, il positionne une variable crédit = 1.
Tous les processus sont initialement passifs
Ils deviennent actif par réception d'un message de travail du leader

Quand le leader envoie un message de travail au processus p, il joint à ce message le réel $c/2$ et conserve un crédit $c/2$

Quand un processus p avec un crédit c_p envoie un message à q, il attache $c_p/2$ au message et conserve $c_p/2$

(invariant: la somme des crédits ds les processus et les messages (transportés par) = 1)

Quand un processus avec un crédit c_p reçoit un message transportant un crédit c'; il fait $c_p := c_p + c'$

Quand un processus devient passif, il renvoie son crédit au leader
Le leader a une variable retour, initialisée à 0 et qui est modifiée par les crédits rendus au leader

Quand retour = 1, le leader annonce la terminaison.

Invariants:
1) La somme de tous les crédits (messages et processus) est égale à 1.
2) Un message de travail a un crédit strictement positif.
3) Un processus actif a un crédit strictement positif

Pseudo-code:

{pour p}
var state$_p$: (actif, passif) : int si p=p_0 donc actif sinon passif;
cred$_p$: fraction init si p = p_0 alors 1 sinon 0;
ret : fraction init 0; { pour p_0 seulement };

Si state = actif → (piggybacking)
envoyer(mes, cred/2); cred = cred/2;

quand un message mess arrive à p →
recevoir(mes, p); state$_p$:= actif;
cred$_p$ = cred$_p$ + c;

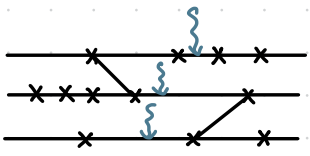
Quand l'état de p passe d'actif à passif →
envoyer(ret, cred$_p$) à p_0;
cred$_p$:= 0;

preuves: si l'application n'est pas terminée, p_0 n'annonce pas la terminaison (fairness)
↳ 1 ps actif (au moins)
→ 0 ps actif mais un mess de travail circule. Ds ce cas, ret ≠ 1

terminée → 0 ps actif, pas de messages de travail. ret $\overset{+c}{=}$ 1.

Quand un message ret arrive à p_0 →
recevoir(ret, c); ret = ret + c;
si ret = 1 alors <Annoncer>;

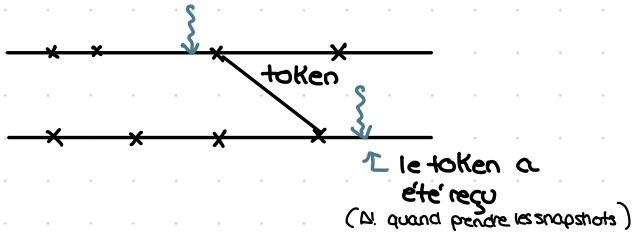
Snapshot / Instantané / Prise d'état global.



À un moment donné, l'état local de l'application est sauvegardé s_i^*
Snapshot: $< s_1^*, s_2^*, \dots >$
↓ configuration

Détection d'une propriété stable (e.g. terminaison, blocage ...)

Hypothèse : l'état d'un processus p contient la liste des messages émis vers le processus q, depuis le début de l'execution $send^*_{p,q}$
et ainsi que la liste de messages reçus du processus q depuis le début de l'execution $recieved^*_{p,q}$



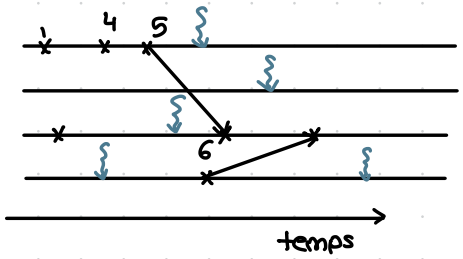
(1) Def: Un snapshot est possible ssi pour tout processus p et q, $send_{p,q} \subseteq recieved_{q,p}$
(l'ensemble des messages envoyés de p à q est inclus dans l'ensemble des messages reçus par q de p)
(→ le schéma ne peut pas se produire)

(2) Def: Une coupe cohérente est un ensemble d'évènements clos par la relation de causalité (α_p)
Relation d'ordre $L_1 \subseteq L_2$: Un snapshot correspond à une coupe maximale pour l'inclusion
éléments préshots
autre éléments post - shot

(3) Def: Un snapshot est significatif si la configuration C* peut être atteinte par une exécution

On va prouver (1)+(2)+(3):
S* est un snapshot possible
e, e' $e \longrightarrow e'$
e' est la réception d'un message émis d'un évènement e
On doit prouver que si e' est pre-shot, e est pre-shot.

S* est une coupe cohérente
On suit une execution qui peut atteindre S*



Tout message pre-shot est reçu dans un evenement préshot.

Cours 11 01/04/2026

Algorithme de Chandy-Lamport

Suppose les canaux FIFO

A la demande de prise de snap shot, le snapshot est pris et un message de type particulier le ' ~~token~~ marqueur ' est envoyé à tous les voisins

justification: les liens sont FIFO

À la réception d'un ~~token~~ marqueur, s'il n'a pas déjà pris son snap shot, le processus le prend et envoie un ~~token~~ marqueur à tous ses voisins

var: token booléen initialisé à faux

À la réception d'un signal de la couche sup:

```
enregistrer l'état local;
    token := vrai
    pourtout q dans voisins
            envoyer <msg> à q
```

Algorithme de Lai et Yang (-)

À chaque message envoyé par l'application, le système a un champ indiquant si le message est preshot ou postshot.

Si un processus reçoit un message (appli) qui est post-shot et si ce processus n'a pas pris son snapshot, il le prend immédiatement et ensuite, reçoit le message.

(-)

var token: booléen init à faux

à la réception de init →

```
enregistrer l'état local;
token := vrai;
```

lors de l'envoie d'un message m (pour l'application) →

```
envoyer(m, token);
```

lors de la réception d'un message <mes, c> →

```
si c et non token alors | enregistrer l'état local
                        | token := vrai;
traiter le message <mes>;
```

Défaillance:

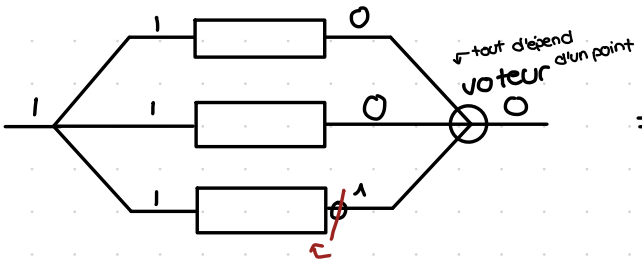
Des processus peuvent mal fonctionner (arrêt total, fonctionnement non conforme à la spécification, malveillance)

Arrêt complet: A un point d'une execution 1 ou plusieurs processus cessent de fonctionner

omissions

Défaillances Byzantines:

Le problème des Généraux Byzantines



⇒ réaliser un "voteur" distribué

Problème : Initialement, chaque processus a une valeur entière
Il s'agit de concevoir un algorithme réparti à l'issue duquel

i) Chaque processus non défaillant décide une valeur (f.mesn)

ii) Deux processus non défaillants ne décident pas des valeurs différentes

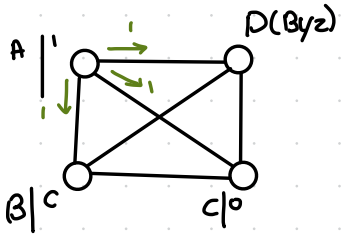
iii) Si tous les processus non défaillants ont la même valeur initiale, cette valeur est la seule valeur de décision possible.

preuves:

(tous les processus corrects choisissent la m valeur)

1) le problème est facile — consensus byzantin
2) le problème est difficile
3) le problème est impossible
4) Solution

2)



1 ou 0 ?

	A	B	C	D
A	1	1	1	
B	1	1	1	
C	1	?	0	0
D	1	0	0	

4) Fisher/Lynch/Saters

Dans un système asynchrone, le problème du consensus est impossible, même avec un seul processus défaillant et un nombre arbitrairement grand de processus corrects.

cas synchrone et défaillances "arrêt total"

flood set : f une borne supérieure au nombre de défaillances ($f < n$)
chaque processus exécute une succession de $f+1$ phases
Initialement un processus initialise un ensemble d'entiers ω à $\omega = \{\text{valeur initiale}\}$;
Chaque processus répète $f+1$ fois:
- envoyer ω à tous
- recevoir les ω_i, et faire $\omega := \omega \cup \{\omega_i\}$;

À la fin de la phase $f+1$,
si $\omega = \{x\}$ alors décider x
sinon décider(0);

Si tous les processus ont la même valeur initiale, alors la seule valeur de décision possible est cette valeur initiale.

Le cas restant est lorsque les valeurs initiales sont différentes.

si à un point de l'execution, les ensembles ω_i des processus sont égaux, $(\omega_j = \omega_i \ j=x)$ alors ils restent égaux jusqu'à la fin du calcul.
Donc tous les processus qui décident décident la même valeur
Il existe au moins une phase au cours de laquelle aucun processus n'est défaillant.
Lors de cette phase, chaque processus reçoit le même ensemble de valeurs.

Lamport, Shosstak, Pease