TD 1 28/01/26

Exo 1: Donnée :

2 processus A et B reliés par un lien bidirectionnel FIFO asynchrone

A et B ne peuvent envoyer que deux messages 0 et 1

3 propriétés :

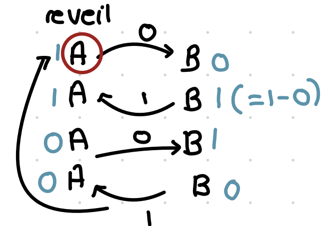
1) A et B envoient chacun une infinité de messages 0 et une infinité de messages 1 ;
2) sur chaque lien orienté, les suites de messages soient toujours de la forme $0^* 1^*$ ou $1^* 0^*$;
3) Chaque processus est toujours en mesure d'envoyer un message.

Q1) Protocoles qui vérifient 2 des 3 propriétés

1 & 2 : variables : x = message

- Lors du reveil envoyer(0)
- répéter toujours : ∅
- lors de la réception de x
  envoyer(1-x)
  si pas reveillé, faire réveil

(reveil : qqun décide qu'on va lancer le protocole)
i.e. démarrer le protocole.
(→ on suppose qu'il y a au moins un reveil spontané, eventuellement plusieurs)



2 & 3 :

- lors du réveil envoyer(0)
- répéter toujours : envoyer(1)
- sur réception de x
  si pas réveiller
  se réveiller
  sinon envoyer (1)

lors de décision d'envoyer : envoyer 0

1 & 3 :

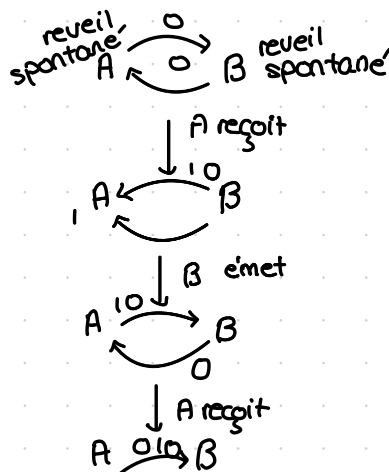
Lors du reveil envoyer(0) envoyer(1)
faire envoyer(0) envoyer(1)

1 & 3 : par b = 0 // ou 1
lors de decision d'envoyer [envoyer b
b = ¬b] (alterne les 1 et les 0)

Q2) Protocole qui vérifie les 3 propriétés :

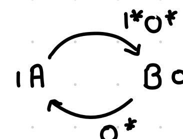
Variables : valeur : entier

- lors du réveil :
  valeur ← 0
  envoyer (valeur)
- repeter toujours :
  envoyer (valeur)
- réception de x :
  si pas réveil alors faire réveil
  A : val ← x
  B : val ← ¬x
  envoyer (val)



Q3) Que se passe-t-il si des liens perdent des messages ?

Même si les canaux perdent des messages, " on tourne " dans l'automate
Si on suppose que c'est B qui ne reçoit jamais 1, mais A en émet une infinité, le canal ne perd pas



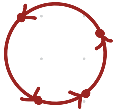
Q4) Interêt du protocole + m̃ propriété avec FIFO, quitte à ce que les messages soient plus gros ?

↳ peu importe si tu perds des messages, la connexion est stable entre les deux. ↳ oui, mais il faut numéroter les messages.

TD 2 04/02/26

Feuille 1, exo 2

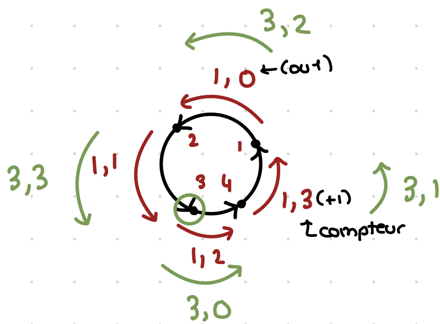
anneau asynchrone orienté



algo réparti pour que chaque processus calcule la taille de l'anneau

2 cas :
- chaque processus a un identifiant unique;
- tous les processus sont identiques

1er cas : Id unique

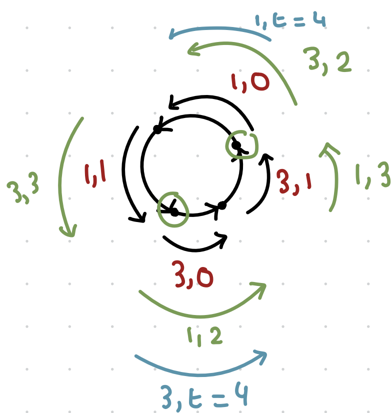


1 reveil spontané

✓ FIFO ou pas FIFO

terminaison locale
(car asynchrone)
complexité : (selon le nb de message) n^2

pour diminuer la complexité (en mieux et en moyenne)



au mieux : un seul reveil : complexité $2n$
(calcul de la taille + diffusion du message)

au pire : $2n^2$

Algorithmes :

(1) Variables. ID : identifiant unique
taille : entier initialisé à -1
asleep : bool init à Faux

Au reveil :
```
asleep ← Faux
var taille = 0
envoyer (ID, 0)
```

Lors de la réception de (ID_reçu, x)
```
si asleep se reveiller
si (ID = ID_reçu) alors
    taille = x+1
sinon envoyer (ID_reçu, x+1)
```

PID : int unique
c : canal (du successeur)
taille : int initialisé à -1
asleep...

un processus au reveil :
```
envoyer (exploration, PID, 0)
asleep False
```

un processus sur reception de (type, id, t)
```
si asleep (asleep faux)
si type = exploration alors
    si id = PID alors
        taille := t+1
        envoyer (diffusion, id, taille+1)
    sinon
        envoyer (exploration, id, t+1)
sinon si type = diffusion alors
    si id = PID alors <terminer>
    sinon
        taille = t
        envoyer (diffusion, id, t)
```

FIFO : terminaison locale si il y a autant de messages d'exploration que de messages de diffusion

↳ pour réduire la taille du message (sans compteur)

(3) sans compteur (si FIFO)

ID : identifiant unique
taille : initialisée à 0
asleep : initialisé à True

ok : bool init à faux.

Au reveil,
  envoyer (ID)
  asleep = False
sur reception de (ID_reçu)
  si asleep, faire reveil
    taille++
  si ID_reçu ≠ ID alors
    envoyer (ID_reçu)

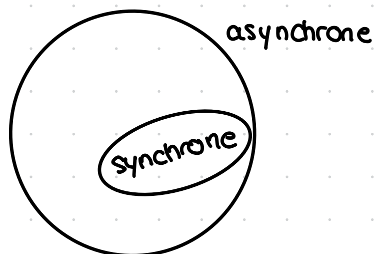
Reveil :
  asleep ← false
  envoyer (mon ID)
sur réception de (ID)
  si ID == mon ID alors
    ok ← TRUE
  sinon
    envoyer (ID)
    si ¬ok alors taille++

cas 2 : tous les processus sont identiques

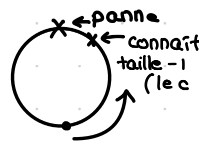
algo impossible

Autres hypothèses :

• si le système est synchrone

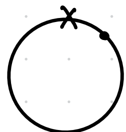


• panne :
→ le dernier noeud connaît taille - 1, mais ne le sait pas



synchrone → $\tau = 1$ ⇝ à la fin $\frac{\text{temps}}{\tau}$
tick ← processus
= nb de ticks d'horloge

• synchrone + panne



m̄ que panne, mais les noeuds savent qu'il y a une panne

• synchrone + bidirectionnel + panne

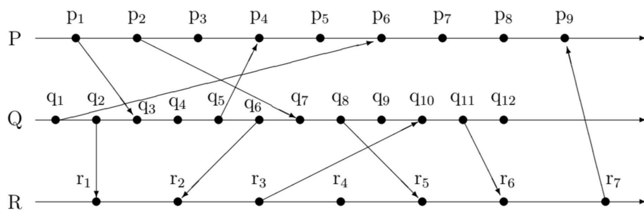
avantages :
• je peux envoyer de chaque côté
• je peux détecter si mon voisin est en panne (mess + acquittement)

1) verif des voisins si pas en panne (mess check et acquittement) ⇝ 4n - 1
2) à tout voisins fiable, envoyer un message de calcul de taille.
3) quand je reçois un mess de calcul de taille, je transfère si mon voisin opposé est fonctionnel
  sinon renvoi à l'envoyeur sous forme mess
4) si je reçois mon retour, j'additionne
  si c'est mon calcul, je stocke (pour savoir s'il faut additionner les 2 côtés)

$O(n^2)$: $4(n-1) - 2 + 2(n-3) \cdot n$

TD 3 11/02/26

TD Horloges : Exo 1 :



lignes de temps en local
mais les lignes ne sont pas comparables

1) que représentent p_6, q_1, p_2, r_4 ?

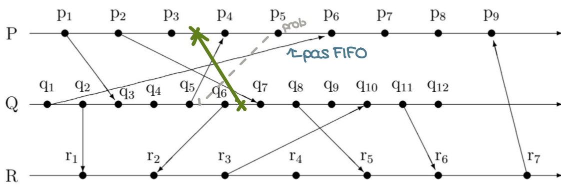
p_6 : réception
q_1, p_2 : envoi
r_4 : evenement interne

types de messages

* on dit que $a \to b$ si :
i) a est avant b sur la même ligne
ii) a est un envoi et b la réception correspondante
iii) il existe une chaîne $a \to \dots \to l$

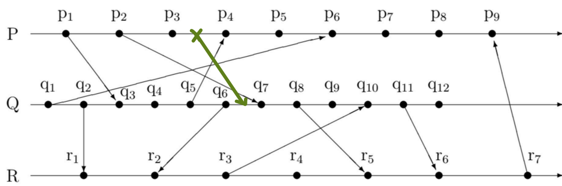
2) Peut-on ajouter :

a) une émission de message juste après q_6 et sa réception juste après p_3 ?



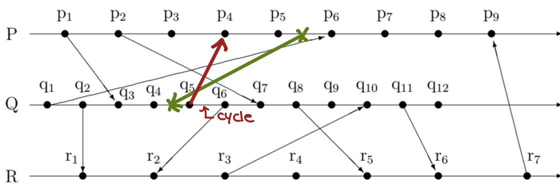
canaux pas FIFO : possible
↳ q_1, q_5
(si pas de cycle → pas de problème de causalité)

b) ______ p_3 ______ q_6 ?



possible

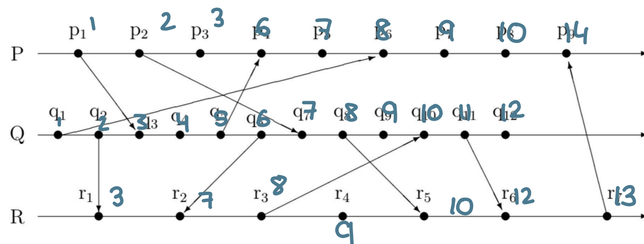
c) ______ p_5 ______ q_4 ?



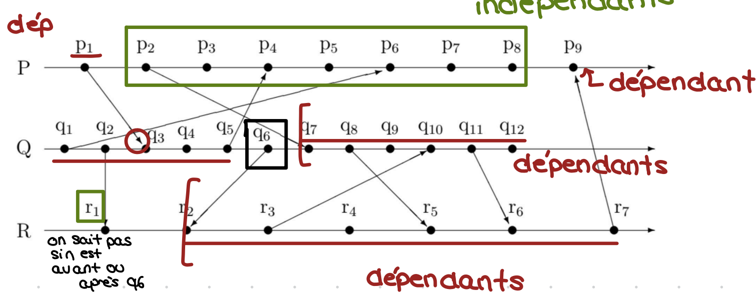
pas possible

3) Calculer les horloges de Lamport :

evn interne / envoi : incrémentation de 1
reception : le max + 1
en P après p_9 : 14
en Q après q_{12} : 12
en R après r_7 : 13



4) Evn indépendants de q_6 ?



5) Mq $L(a) < L(b) \not\Rightarrow a \rightarrow b$

Ex : p_3 et q_4

$3 < 4$ or p_3 et q_4 sont indépendants

ou q_6 et p_8 (indépendants)

6) D'après le diagramme, les voies de communication préservent-elles l'ordre des messages ?

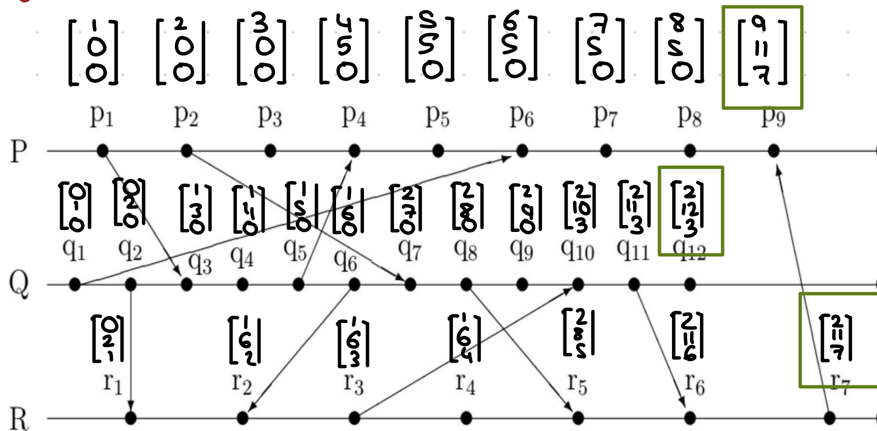
<u>NON</u> pas FIFO (message envoyé en q_5 reçu avant celui en q_1)

5) Z_{min} : min temps d'envoie de message (on néglige les evnts locaux)

algo : calculer la durée min de l'algo décrit par la relation de causalité ?

(Lamport en ignorant les évènements internes et les réceptions, ⇝ prende la valeur max et la multiplier par Z_{min}.) max(clock de chaque process) $* Z_{min}$

8) Horloge de Matter



1) (0,0,0) devant chaque processus

2) incrémenter i $(i+1)$ pour chaque evnt interne

3) max de <u>l'estampille</u> (le reste) si émission ou evnt local de l'evn précédent (ou précédent + celui qui a émis le message si réception)

<u>vecteurs</u> : une entrée par processus

↳ un elt de b > elt de a

9) Mq $M(a) < M(b) \Leftrightarrow a \rightarrow b$

preuve :

(⇐) par définition

(l'evnt qui est causalement après va augmenter l'une des composantes du vecteur)

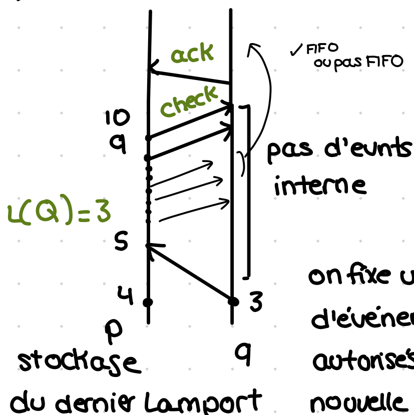
(⇒) On suppose $M(a) < M(b)$

donc en particulier $M(a)\big|_{p(a)} < M(b)\big|_{p(a)}$ où $p(a)$ est le processus sur lequel a lieu a

un événement de b est plus grand pour le process $p(a)$. (b a "vu" au moins $M_{p(a)}(a)$ evenements de $p(a)$, donc a est dans le passé causal de b : $a \rightarrow b$

Exo2 :

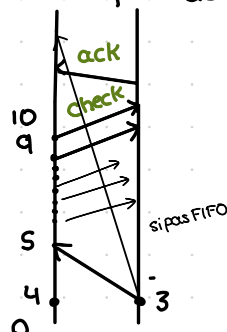
1) algo reparti à deux processus pour borner la valeur absolue de l'écart entre leurs deux horloges



on fixe un nb d'évènements / envois autorisés sans nouvelle preuve que l'autre a avancé.

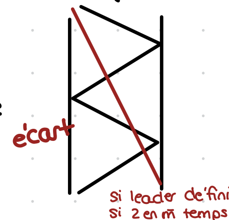
2) L'écart entre la valeur sur un processus et sur un message est-il aussi borné ?

Oui (P n'envoi pas de message tant qu'il a pas reçu l'aquittement



→ sol possible

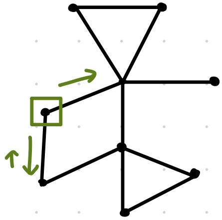
on acquitte à chaque fois



3)
pour généraliser ?
relations 2 à 2 à chaque voisin

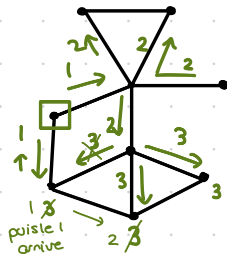
4) Pourquoi cela peut être utile ?
diminuer la taille des messages
+ reset au bout d'un moment (l'horloge peut compter indéfiniment)
limite le nb de bits qu'on envoi

TD : Algo création d'arbre



synchrone : ✓ arbre en largeur (fixe)
asynchrone : fil
(pas forcément un arbre en largeur)
↳ pour les diffusions d'information

↳ on rajoute une notion de 'distance'



⇝ profondeur 3

→ pour la terminaison
diffuser
⇝ puis répondre oui
(pas l'inverse

complexité :
au pire : nb arcs x nb sommets

TD 4 18/02/26

(Feuille de TD 1)

Exo 3 :

réseaux liens bidirectionnels + processus distingué

1) protocole de diffusion d'informations par le processus distingué

booléen deja_vu (sinon on ne s'arrête jamais)
plusieurs info :
hash / idée

pour tout processus i

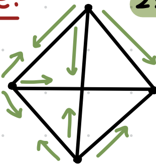
variable dejaVu : bool à faux

fct Reveil ( ) : si je suis le leader
dejaVu ← vrai
envoi (Info) à tout mes voisins

lors de reveil Reveil ( )

lors de réception de Info sur c :
si ¬ dejaVu alors
dejaVu ← Vrai
envoyer (Info) à tous les voisins (sauf c)

complexité : 2x. le nb d'arrêtes (borne supérieure)



2) avec les acquittements

pour tout processus i

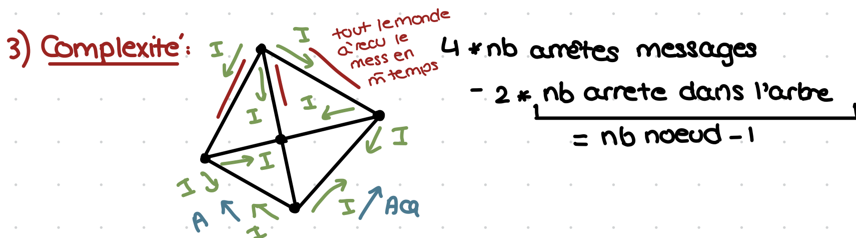
variable pere : canal
dejaVu : bool à faux
acqVoisins : entier — liste ou nb de voisins ↳ « messages frauduleux »

lors de reveil si je suis le leader
dejaVu ← vrai
envoi (Info) à tout mes voisins
acqVoisins ← mon Degré (nb d'acq que je dois recevoir)
sinon
acqVoisins ← mon Degré - 1
pere ← c
si acqVoisins == 0 alors envoyer (Acq) à pere
(pas nécessaire)

lors de réception de Info sur c :
si ¬ dejaVu alors
dejaVu ← Vrai
envoyer (Info) à tous les voisins sauf c
acqVoisin ← monDegré - 1
sinon
envoyer (Acq) à c

Lors de réception de (Acq) sur c :
acqVoisin --
si acqVoisins == 0 alors si non Leader envoyer (Acq) à pere
sinon <terminer>

3) Complexité :

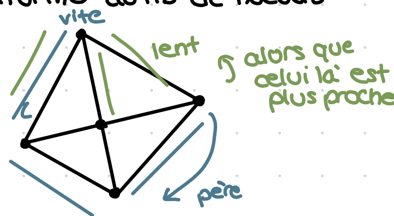


4 * nb arrêtes messages
- 2 * nb arrete dans l'arbre
= nb noeud - 1

4) Complexité en temps

temps fixe τ : = 2 * le rayon centré au leader × τ

transmission borné par τ : au pire : arbre filiforme au nb de noeuds
= 2 * nb noeuds × τ



→ Création d'un arbre :

algo distance, algo vague

algo distance :

sans accusé de réception :

var : pere : canal
dist : entier à ∞

Au reveil :

si proc distinguer alors
envoyer(distance (1)) à tous les voisins
dist = 0

sur reception de distance(x) sur c :

si x < dist alors
pere ← c
distance ← x
envoyer distance(x+1) à tout les voisins sauf c

avec accusé de réception :

var : pere : canal
dist : entier à ∞
AcqVoisins : entier à 0

Au reveil :

si proc distinguer alors
envoyer(distance (1)) à tous les voisins
dist = 0
AcqVoisin = monDegré

sur reception de distance(x) sur c :

si x < dist alors
~~pere ← c~~
si pere ≠ Null alors
envoyer (Acq)
pere ← c
distance ← x
envoyer distance(x+1) à tout les voisins sauf c
AcqVoisins += monDegre - 1 (~ les prochains doivent avoir la bonne distance)
si AcqVoisins == 0 alors envoyer Acq à père
sinon // si x ≥ dist
envoyer (Acq) à c

sur reception Acq de c
AcqVoisins --
si AcqVoisins = 0 alors
si proc distingué <terminé>
sinon
envoyer Acq à père

complexité :

au pire : 2x nb de sommets x nb d'arrêtes (nb messages)

$2 \cdot |S| \cdot |A|$

$|V| \cdot |E|$

$O(|S| \cdot |A|)$

en temps : $O(rayon)$

algo des vagues : (voir cours
version avec acquittement)

complexité :

en temps : $O(rayon^2)$ 1, 2, 3 … somme de 1 à rayon

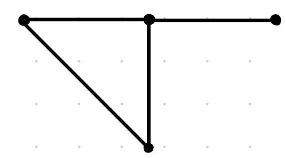
en message :

$O(rayon \times |S| + A)$

(écrire l'algorithme + complexité)
+ex4.
entrainnement.

algo vague [pas fait en TD] (asynchrone)

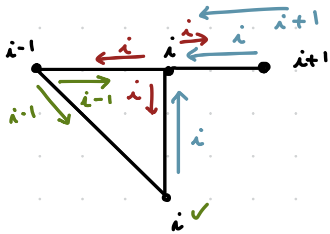
TD 5   11/03/2026

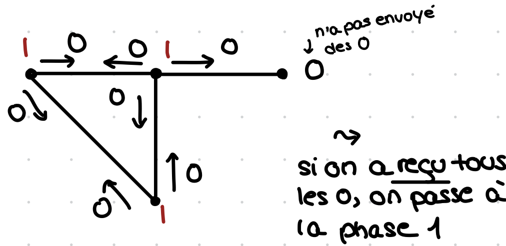


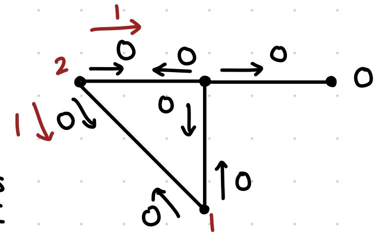
Algorithmes de construction d'arbre

synchroniseur: (executer un algorithme synchrone sur un réseau asynchrone)
Une phases d'écart entre voisins

nb phases ≤ diamètre







pour simplifier les messages: booléens au lieu d'entiers (pas d'écarts ≥ 1)

Pseudo-code:

var flipflop = vrai
Var Mess [voisins, bool]: initialisé à vide partout
Var état = endormi
var père = indéfini

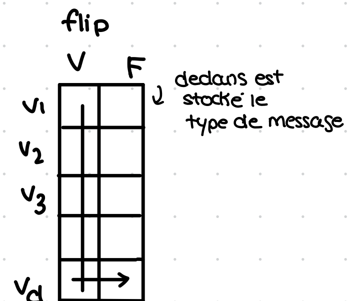
Reveil

  état = réveillé
  si leader alors
      envoyer (arbre, flipflop) à tous les voisins
  sinon envoyer (RAS, flipflop) à tous les voisins
      (↳ rien à signaler)

Lors de reveil spontané Reveil()

Lors de réception (info, b) sur c

  si état = endormi alors Reveil()
  si pere ≠ indefini alors ARatt --
      si ARatt == 0 alors
          envoyer(AR) au père



  Message [c, b] = info
  si pour tout Message [c, flipflop] ≠ vide
    si pour tout les voisinss v Mess[v, flipflop] == RAS
        envoyer (RAS, flipflop)
        pour tout v, Mess[v, flipflop] = vide
        flipflop = not(flipflop)
  sinon
    pere = id le plus plus grand parmi les noeud pour lesquels Message [id, flipflop] = arbre
    (accusé de réception)
    ARatt = nbvoisins
    pour tout voisin v
        si Mess [v, flipflop] = arbre alors
            ARatt -- (lui aussi il voulait être père)
            si pere ≠ v alors envoyer AR à v
        sinon
            envoyer (arbre, !flipflop) à v
    fp

Si AR att == 0 alors envoyer AR à pere
flipflop = ! flipflop

fin des enfants / des autres qui ne sont pas dans mon arbre

si pas FIFO ?

(Feuille de TD n°2)

Exclusion mutuelle :

Réseau connexe

Noeud en anneau

Exo 1 :

Algorithme d'exclusion mutuelle de Lamport

var horloge = 0
var Date [degré] d'entiers initialisé à ∞
var état ∈ { neutre, SC, demande } init à neutre

sur volonté de passer en SC :

état ← demande
horloge ++
envoyer ("demande", monId, horloge) à tous
Date [monId] ← horloge

sur réception (type, Id, h)

Date [Id] = h
horloge = max(h, horloge + 1)

si etat = neutre ET type = demande alors
envoyer ("ok", mon ID, horloge)

si etat = demande et pour tout v Date([monId], monId ≤ (Date[v], v)
etat = SC
je rentre en SC

(si jamais 2 ont la m̂ horloge)

sur sortie section critique

etat = neutre
horloge ++
Date [monId] ← ∞
pour tout v :
envoyer ("ok", mon_Id , horloge)

(algo de Ricart et Agrawala dans la feuille)
tampon: historique (de taille finie)
(la dernière SC est à telle date)
il faut tout parcourir (boucle)

TD 6 18/03/2026

1) Mq l'algo de Ricart et Agrawala ne respecte pas $a \to b \Rightarrow$ (a est satisfaite avant b.)
non car ça dépend de si le processus qui a le jeton reçoit la demande de a avant celle de b

2) Comparer les taux moyen d'occupation de la s.c.
suppose que :
- $t_{transmission} = \beta + \tau L$ (L = longueur moyenne en octet)
- T^* = temps d'occupation moyen de la s.c.
- estampilles = 4ϕ, id = 2ϕ

taux d'occupation = $\frac{\text{temps en S.C}}{\text{temps en SC et hors SC}}$

hors SC : temps de transmission du jeton
$\beta + \tau L$
$L = 4n + 1$ (1 : type de message)

A.N : $\tau = 0.1\ ms/\phi$
$\beta = 10\ ms$
$n = 1000$ $T^* = 1s.$

$\beta + \tau L = 410,1\ ms$

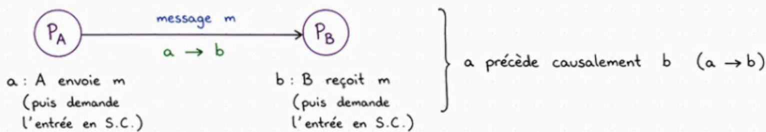
taux d'occupation $= \frac{1000}{1000 + 410} \simeq$ **0,71**

Lamport :
$L = 1 + 2 + 4$ (1 = type de la demande, 2 = mon_id, 4 = estampille)
$\beta + \tau L = 10.7\ ms$
$\frac{1000}{1000 + 10.7} =$ **0.99**
↑ avec l'hypothèse d'enchaînement

Exclusion mutuelle : causalité et ordre des entrées en S.C.

Question : si un événement a (demande d'entrée d'un processus A) précède causalement un événement b (demande d'entrée d'un processus B) (a → b), alors est-ce que A est forcément satisfait (entre en S.C.) avant B ?

1) Algorithme de Ricart - Agrawala



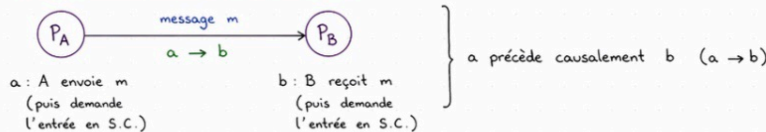
a précède causalement b (a → b)

Conclusion :
L'algorithme de Ricart - Agrawala ne garantit pas que A entre avant B.
Donc a → b n'implique pas A est satisfaite avant B.

Pourquoi ? Quand B diffuse sa demande, elle peut avoir une estampille plus petite que celle de A (car B a reçu m avant d'incrémenter son horloge).
Donc : B peut avoir le privilège et entrer en S.C. avant A, même si a → b.

---

2) Algorithme de Lamport (centralisé)



a précède causalement b (a → b)

Conclusion :
L'algorithme de Lamport garantit que A est satisfaite avant B.
Donc a → b implique A est satisfaite avant B.

Pourquoi ?
- A diffuse sa demande avant b.
- B ne peut pas diffuser sa demande (et donc obtenir une estampille plus petite) avant d'avoir reçu le message m de A (causalité).
- Comme les canaux sont FIFO, tous les processus reçoivent d'abord la demande de A.
- Donc la demande de A est forcément classée avant celle de B dans toute les files d'attente.
- Ainsi, A est la plus petite et entrera avant B.

Rappel :
Dans Lamport, l'entrée se fait uniquement si sa demande est la plus petite dans tous les tableaux des processus.

Résumé :
- Ricart - Agrawala : a → b n'implique pas A avant B.
- Lamport (centralisé) : a → b implique A avant B.

Intuition : Lamport impose un ordre global cohérent avec la causalité ;
Ricart - Agrawala n'impose que l'ordre parmi les demandes concurrentes.

Exo2:

9 Considérons l'algorithme de Naimi et Trehel.

état initial du système :

	S_1	S_2	S_3	S_4	S_5	S_6
Dernier	1	1	1	1	1	1
Suivant	0	0	0	0	0	0
Racine	V	F	F	F	F	F
Privilège	V	F	F	F	F	F
Demande	F	F	F	F	F	F

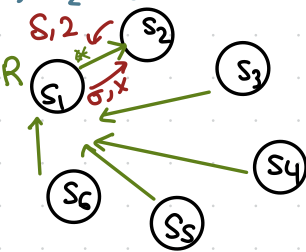
```
lors de la décision d'entrer en section critique faire
  début
    demande ← vrai
    si non racine alors
      envoyer (δ,mon_num) à dernier
      racine ← vrai
      dernier ← mon_num
    fin si
    attendre privilège
    entrer en section critique
  fin
```

```
lors de la sortie de section critique faire
  début
    demande ← faux
    si non racine alors
      envoyer (σ,indéfini) à suivant
      suivant ← 0
      privilège ← faux
    fin si
  fin
lors de la réception d'un message m faire
  début
    si Π1(m) = σ alors privilège ← vrai
    si Π1(m) = δ alors
      si racine alors
        racine ← faux
        si demande alors suivant ← Π2(m)
        si privilège et non demande alors
          privilège ← faux
          envoyer (σ,indéfini) à Π2(m)
        fin si
      sinon envoyer m à dernier
      fin si
    dernier ← Π2(m)
    fin si
  fin
```

(Algo de Naimi et Trehel)

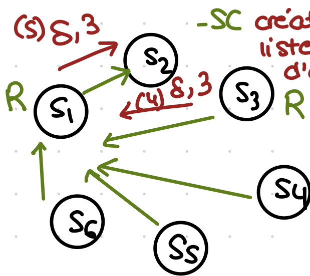
Celui qui a fait la demande la plus récente devient la racine.

(1) S_2 demande d'entrer en S.C +(2) +(3)



	S_1	S_2	S_3	S_4	S_5	S_6
Dernier	2	2	1	1	1	1
Suivant	0	0	0	0	0	0
Racine	F	V	F	F	F	F
Privilège	F	V	F	F	F	F
Demande	F	V	F	F	F	F

(4) (5)(6) S_3 demande, S_1 reçoit la demande de S_3, S_2 reçoit le message de S_1

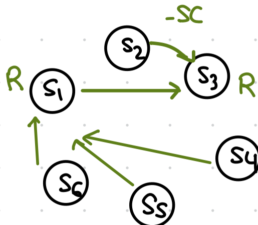


-SC création de la liste d'attente

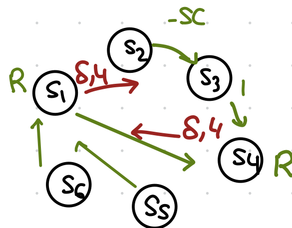
	S_1	S_2	S_3	S_4	S_5	S_6
Dernier	2	2	3	1	1	1
Suivant	0	0	0	0	0	0
Racine	F	V	V	F	F	F
Privilège	F	V	F	F	F	F
Demande	F	V	V	F	F	F

S_2 reçoit :

	S_1	S_2	S_3	S_4	S_5	S_6
Dernier	3	3	3	4	1	1
Suivant	0	3	4	0	0	0
Racine	F	F	V	F	F	F
Privilège	F	V	F	F	F	F
Demande	F	V	V	F	F	F

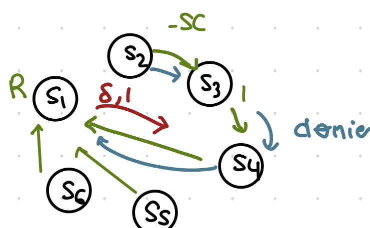


(6) S_4 fait une demande



	S_1	S_2	S_3	S_4	S_5	S_6
Dernier	4	3	4	4	1	1
Suivant	0	3	4	0	0	0
Racine	F	F	F	V	F	F
Privilège	F	V	F	F	F	F
Demande	F	V	V	F	F	F

S_1 envoi à S_4 :



	S_1	S_2	S_3	S_4	S_5	S_6
Dernier	1	3	4	1	1	1
Suivant	0	3	4	1	0	0
Racine	V	F	F	F	F	F
Privilège	F	V	F	F	F	F
Demande	F	F	V	V	F	F

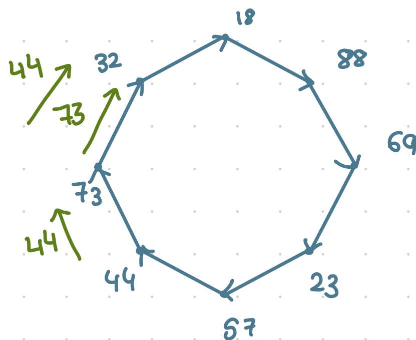
	S_1	S_2	S_3	S_4	S_5	S_6
Dernier	1	3	4	1	1	1
Suivant	0	0	4	1	0	0
Racine	V	F	F	F	F	F
Privilège	F	F	V	F	F	F
Demande	F	F	V	V	F	F

2) topologie: arbre à l'envers

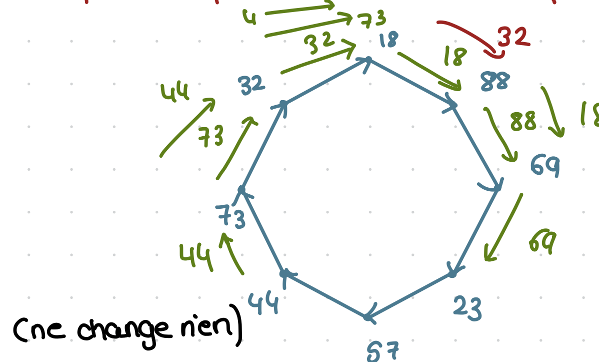
3) n-1

4) Complexité en nb de messages
meilleur cas: à distance 1 (profondeur 1)
pire cas: SC longue + beaucoup de demandes pendant cette SC longue
arbre filiforme
profondeur log(n)

Feuille de TD3:
1) Algorithme de Le Lann
(anneau unidirectionnel)



2) Si plusieurs processus se reveillent en parallèle



(ne change rien)

3) Doit être FIFO
4) Complexité en message: n^2 (le max fait le tour + les ids de chaque processus)
5) Complexité en temps au pire: 2n - 1
1 seul reveil: n-1 messages + n pour que son propre message arrive à lui-même
6) Algorithme filtré

10 Considérons l'algorithme d'élection de Le Lann (1977) valide sur un anneau unidirectionnel :

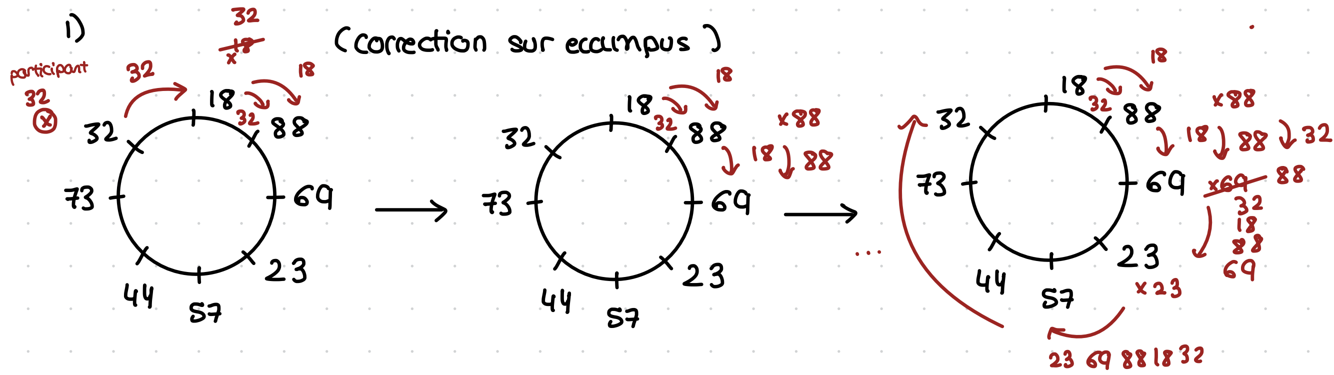
lors du réveil
  début
    participant ← vrai
    max ← mon_numéro
    envoi(élection,max)
  fin

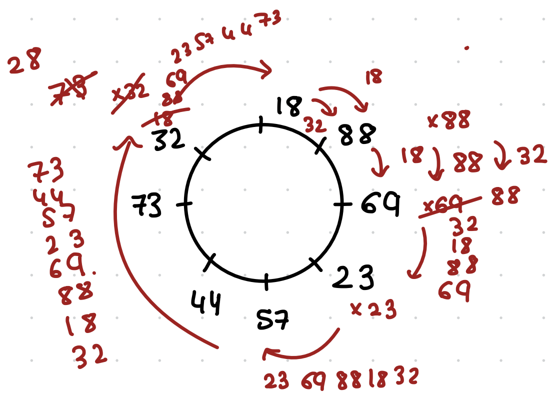
lors de la décision de lancer une élection
  réveil

lors de la réception de (élection,j)
  début
    si non participant **alors** réveil
    si mon_numéro = j **alors**
      élu ← max
      participant ← faux
    sinon
      si j > max **alors** max ← j
      envoi(élection,j)
    fin si
  fin

TD 7 25/03/26 (Feuille 3)

1) (Correction sur ecampus)





incrémente le max jusqu'à 88
chaque num reçu après n'est plus renvoyer
tout le monde voit passer tous les id

3) FIFO?

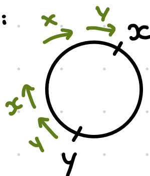
oui : si c'est pas Fifo :

si le max 'traine' ds le canal alors que tous le reste circule.

(pour l'algo en pas fifo)
astuce : si on a tout reçu : chacun connait l'id du prédecesseur

on reçoit (id, id_prec) : un id apparait comme id_pred mais pas comme id principal

si c'est Fifo :



4) <u>Complexité au pire :</u> $O(n^2)$

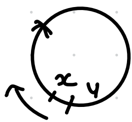
n processus. chacun reçoit n Id.

5) <u>au temps au pire</u>

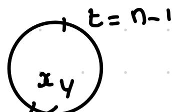
(moins d'une unité de temps)

2n :

t = 0

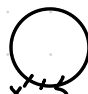


t = n-1



reveil provoqué

t = n-1 + n



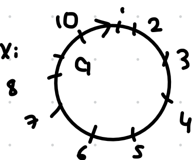
le message de y fini le tour

6) <u>filtrer les messages</u>

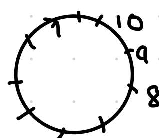
indentation (envoi (election, j)) (mettre dans le si)

<u>complexité au pire :</u> si les numéros sont dans l'ordre décroissant par rapport au sens de l'envoi

mieux :



<u>pire :</u>



1 fait 1 pas
2 fait 2 pas
...
10 fait 10 pas
n fait n pas

$\frac{n(n-1)}{2}$, $O(n^2)$

7) FIFO? pas nécessairement que le process gagnant arrive à faire le tour

8) ids répartis aléatoirement.
(au pire : triés décroissants)

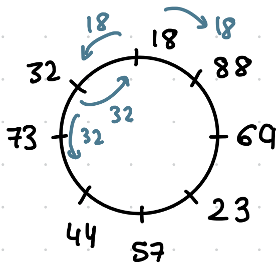
max : distance n
2e max : n/2 en moyenne
3e max : n/3 (pour raisons de symétrie)
...
Ke max : n/K

$\Rightarrow$ total : $n \cdot \sum_{i=0}^{n} \frac{1}{i} \sim n \ln(n)$

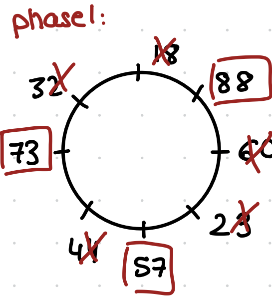
↝ comment avoir $O(n \ln(n))$?
redéfinir les ids par une fonction de hachage.

Exo 2 :
élection sur un anneau bidirectionnel

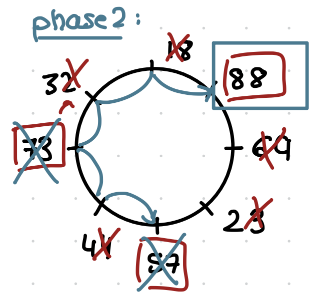
1) • chaque p doit connaître ses voisins (au réveil)



on meurt si voisin >
phase 1 :

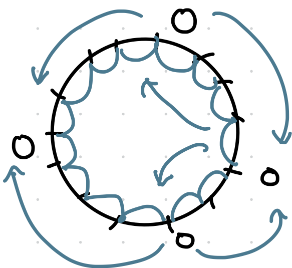


phase 2 :



phase 3 : 88 reçoit son propre message
↝ 88 est élu.

à chaque étape, le nb de processus vivants λ est, au moins, $\lambda/2$.
un processus reste vivant si ses 2 voisins meurts : 2 x plus de morts que de vivants
/!\ on ne compte pas les process 2 fois
nb phases : $\leq O(\log_2(n))$
nb messages : 2n



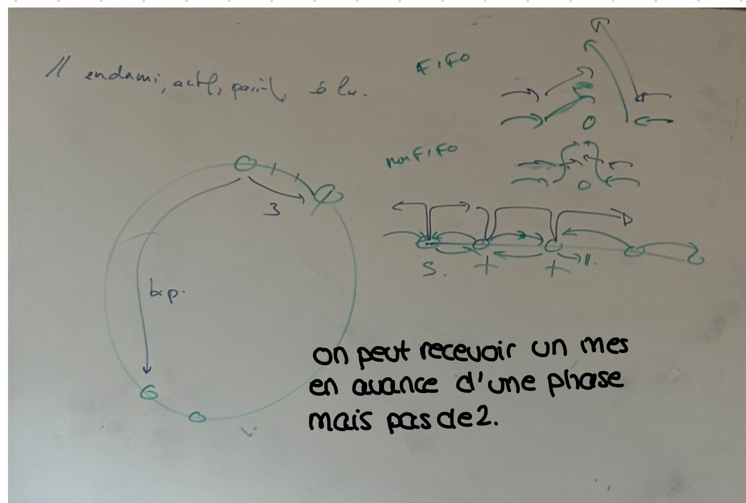
en moyenne :
$\mathbb{P}$(process reste vivant à la fin de la phase 1) = 1/3. (ou : 3 valeurs aléatoires. 1/3 proba que le max soit au milieu
6 possibilités. 2 où ils survient

$\Rightarrow O(\log_3(n))$

3) code :

etat ← endormi // endormi, actif, passif, élu
// phases pas synchrone bcp (3)
phase ← vrai
T[cote , phase] = vide partout // contient des ids

reveil : etat = actif
envoi (mon id, vrai) aux 2 voisins;



reception de (id, p) sur c :
si etat = passif
alors [envoi(id, p) sur !c
break
si etat = endormi alors reveil ( )
T[c,p] = id
si T[G, phase] ≠ vide et T[D, phase] ≠ vide // pas de sinon
alors | si T[G, phase] == monId // et D droit
alors etat = elu
sinon si T[G, phase] > monId ou T[D, phase]
alors etat = passif ; phase = non phase
// si on a reçu de messages en avance de phase
pour c ∈ G, D
si T[c, phase] ≠ vide }
alors envoyer T[c, phase] à !c
sinon | pour c ∈ G , D T[c, phase] = vide
phase = non phase
envoyer(monId, phase) des 2 côtes
si c'est FIFO end of code
sinon [(souligné)

4) anneau unidirectionnel :
que d'un coté :
complexité au pire :
nb phases : n (triés croissants → 1 qui meurt à chaque phase)
au mieux : triés décroissant

Si on alterne : les grands tuent les petits en phase paire
l'inverse en phase impaire.
⇒ $\lesssim 2\log_2(n)$ nb phases
complexité : $n . 2\log_2 n$
(ou : 'simuler' l'algo bidirectionnel . meurt si <
que l'un des 2

3)
en synchrone : 2 bits
l'id le plus rapide est élu
si $t = 2^n$
vitesse $1/2$ id
id + 1 : vitesse 2 fois plus lent
(ne fait que la moitié de l'anneau
↓
3ème : 4 x plus lent . 1/4 de l'anneau
⋮ ... ⇒ nb messages : $n + n/2 + \frac{n}{4} + \frac{n}{16} = 2n$
linéaire.

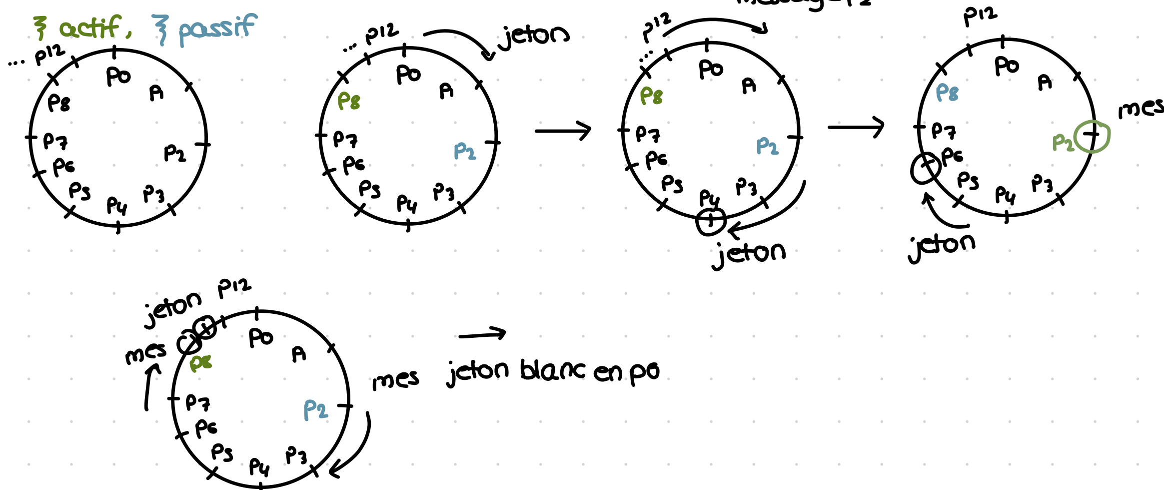
TD 8 01/04/2026

Exo 1 :

- réseau complet asynchrone
- proc passif : ne fait pas de calcul tant qu'il n'a pas reçu de nouveau message
- terminaison : tous les processus sont passifs et aucun message ne circule.

Dire si ce protocole est correct :
anneau unidirectionnel $p_1, \ldots, p_n$
proc passif → transmission d'un jeton blanc
proc actif reçoit un jeton → jeton noir → tout site recevant le jeton noir le transmet noir

Protocole faux : si on considère cette exécution :



<u>étapes</u> : p8 est actif → p0 envoie le jeton → le jeton passe par p2 qui est passif → le jeton arrive en p4 → p8 finit et envoie un message p2, p8 devient passif → le jeton est en p6, le message de p8 arrive en p2 qui devient actif le jeton passe par p8, qui est passif, le jeton passe en p10, p2 finit et envoie un message p8, p2 devient passif → le message de p2 arrive en p8 qui devient actif, le jeton est en p12 → le jeton arrive blanc en p0

→ Il y a toujours de l'activité et le jeton n'a rien vu
le déroulement peut être itéré à l'infini

Exo 2 :

var
couleur: {blanc, noir} **initialisé à** noir
état: {actif, passif} **initialisé à** actif
jeton_présent: booléen **initialisé à** (mon_num == 1)
nb: entier

lors de

réception de (message,m)
début
état ← actif
couleur ← noir
fin

attente
début
état ← passif
fin

réception de (jeton,j)
début
nb ← j+1
jeton_présent ← vrai
si nb == taille **et** couleur == blanc
alors terminaison détectée
fin si
fin

jeton_présent **et** état == passif
début
si couleur == noir
alors nb ← 0
sinon nb ← nb+1
fin si
envoyer (jeton,nb) **à** successeur
couleur ← blanc
jeton_présent ← faux
fin

1) qu'indique le fait d'être blanc /noir ?

un process est blanc s'il n'a pas été actif depuis la dernière fois qu'il a vu le jeton.
il est noir s'il n'a jamais vu le jeton ou a été actif depuis la dernière fois qu'il a vu le jeton.

2) Combien de tours au maximum le jeton peut-il parcourir entre la terminaison et le moment où elle est détectée ?

2 tours : à l'instant T, il n'y a plus d'activité mais certains process sont noirs car ils ont été actifs récemment.
Il faut un tour de jeton pour remettre tous les process à blanc, et un deuxième tour pour le jeton parvienne à compter jusqu'à taille.
cas extrême où il faut effectivement 2 tours : l'activité a lieu au process pX juste derrière le jeton, le process pX est noir ; puis tout activité s'éteint, il faut bien un tour pour remettre pX à blanc puis pour que pX détecte la terminaison.

3. Quel est le processus qui détecte la terminaison ?

   - Le premier process derrière le jeton qui est noir au moment où l'activité se termine.
     Ce n'est pas a priori le dernier process où il y a de l'activité. Exemple :
     jeton en p4, activité en p2
     jeton en p6, p2 envoie un message à p12 et p2 s'éteint, p2 est noir.
     jeton en p8, p12 reçoit le message de p2 s'active, p12 est noir
     jeton en p10, p12 s'éteint, il n'y a plus d'activité
     le jeton passe en p12 qui est noir, le compteur est rebooté, p12 devient blanc
     le jeton passe en p2 qui est noir, le compteur est rebooté, p2 devient blanc
     le jeton fait un tour, revient en p2 qui détecte la terminaison, ce n'est pas qui fût le dernier actif, c'est p12.

4. Donner des éléments de preuve de cet algorithme.

   - S'il y a terminaison, alors il y a détection (cf. au dessus, il a été expliqué que cela prenait moins de 2 tours).

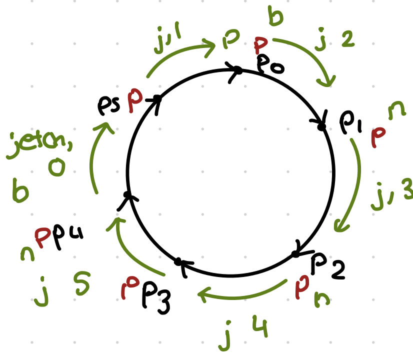
   S'il y a détection au temps T1 au process pD. À T2=T-n, le process pD a envoyé le jeton avec le compteur n. Puis le jeton a fait le tour de l'anneau et n'a vu que des process blanc.
   Un process a-t-il pu être actif à T2 ? Non car dans ce cas, il aurait été noir à T2, et aurait toujours été noir lors du dernier passage du jeton, le compteur d'icelui aurait donc été rebooté et pD n'aurait donc pas reçu le jeton avec le compteur n, donc n'aurait pas détecté la terminaison.
   Un message pouvait-il être en transit à T2 ? C'est ici que nous utilisons l'hypothèse synchrone. À T3=T2+1, le process pX juste après pD détient le jeton, et le message est arrivé à destination, car il met [moins de] 1 pour arriver. Donc à T3, il y a un process noir et le jeton doit encore voir tous les process, il ne pourra donc pas finir son tour tout en ne voyant que des process blancs.

5. Pouvez-vous adapter cet algorithme au modèle où le temps de transmission est borné par une constante T connue ? Et au modèle asynchrone avec des canaux FIFO ?

   - Reprenons le raisonnement précédent, légère variant : T2, c'est le moment où le process pD a envoyé le jeton qui va réussir son tour blanc. T3, c'est le moment où pX envoie le jeton.
     Il faut garantir qu'aucun message n'est en transit à T2.
     Si le temps de transmission est majoré par 1, il suffit que pX attende 1 avant de propager le jeton et le raisonnement ci-dessus tient. Notez qu'il n'est pas utile que tous les process attendent, seulement pX, i.e. celui qui fait passer le compteur de 0 à 1.

     Asynchrone mais FIFO : méthode bourrin : Quand un process P reçoit le jeton, avant de la propager, il envoie "ping" à tous les autres process, lesquels répondent "pong" à la réception de ping. Quand P a reu̧ tous les pong, il est autorisé à propager le jeton.
     S'il y a un message de R vers S à T2, quand le jeton va arriver en S, celui-ci va envoyer ping à R qui renverra pong à S. Mais puisque les canaux sont FIFO, ce pong arrivera après le message en transit, qui remettra S à noir, et donc le jeton va être rebooté avant d'être envoyé par S

Exo 2 :



1) un processus est blanc une fois qu'il devient passif
noir quand il a reçu un message

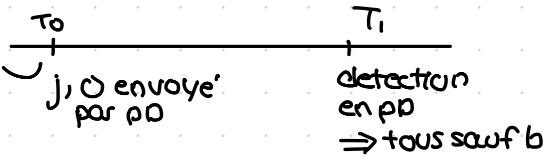
2) 2 tours : 1 → tous les jetons à blanc
2 → pour avoir tous les blancs

3) le dernier processus noir

15 Initialement, tous les processus sont passifs. Le process p_0, le "leader" se réveille spontanément une fois et une seule, il devient alors actif. Les autres process ne peuvent pas se réveiller spontanément. Ils ne peuvent devenir actif qu'à la réception d'un message. Un process actif peut envoyer des messages. Quand il n'a plus rien à faire, il redevient passif et ne pourra redevenir actif que s'il reçoit un message. Ce type de calcul est appelé calcul diffusant. Un tel calcul est terminé si tous les process sont passifs et qu'aucun message ne circule. Nous supposons le temps de transmission d'un message borné par Δ, cette constante est connue.

```
var
  état: {actif,passif} initialisé à passif
  nbfils: [0, n − 1] initialisé à 0
  père: [0, n − 1] ∪ {⊥} initialisé à ⊥
  délai_écoulé: booléen initialisé à vrai

lors de réveil (une fois pour le leader p0)
  état ← actif

lors de attente
  état ← passif

lors de désir d'émettre message(m) vers pj
 début
  envoyer message(m,mon_id) à pj
  délai_de_garde(2Δ)
  délai_écoulé ← faux
 fin

lors de signal d'horloge
  délai_écoulé ← vrai

lors de réception de message(m,j)
début
  état ← actif
  si père == ⊥ alors
    père ← j
    envoyer contrôle(rentrer) à pj
  fin si
fin

quand état == passif et nbfils == 0 et délai_écoulé
début
  envoyer contrôle(sortir) à père
  père ← ⊥
fin

lors de réception de contrôle(m)
début
  si m == rentrer
    alors nbfils ← nbfils+1
    sinon // m == sortir
        nbfils ← nbfils-1
  fin si
fin
```

4) terminaison ⇒ détection
↳ 2N mess (2 tours)
détection ⇒ terminaison



j, 0 envoyé par p0

détection en p0
⇒ tous sauf b

5) borné : attendre un délai ⩾ T avant de considérer un processus passif
asynchrone avec FIFO : Djikstra-Scholten

Exo 3 :

1) À quoi sert 2Δ ?

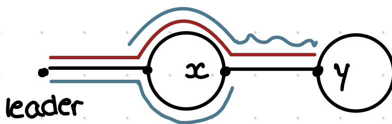
(sert qu'un processus ne quitte trop tôt l'arbre de contrôle alors qu'une activité qu'il a déclanchée n'a pas encore été prise en compte par les messages de contrôle)

Un message envoyé à t arrive au plus à t + Δ

Le message de contrôle associé (rentrer) revient au plus tard à t + 2Δ

2)

phase 1 : leader envoi deux calculs vers x



- Le calcul du haut va construire un arbre qui passe de leader vers x puis y en passant pas le haut.

  Puis le calcul du bas atteint x alors que x est encore dans l'arbre avec père de x dans la partie haute du cercle de gauche. x ne change pas de père. L'arbre du bas s'arrête juste en amont de x.

  Le calcule du bas se propage dans la partie droite, il vient du bas, mais x garde comme père son voisin gauche haut. L'arborescence de leader vers x se maintient en haut pour mémoriser que du calcul se fait à droite alors que ce calcul à droite est arrivé par le dessous à gauche.

1. À quoi sert l'attente d'un délai 2Δ ?

   - Précision d'abord sur le code :

     Délai de garde de 2Δ, cela signifie que nous lançons un minuteur qui sonnera dans 2Δ, si le minuteur était en train de tourner, il est rebooté à 2Δ. Cela veut donc aussi dire que le minuteur sonnera dans 2Δ, à moins qu'il ait été rebooté avant.

     Le protocole construit un arbre dans lequel sont maintenus les process actifs, un process rentre dans l'arbre en envoyant un message "rentrer" à celui qu'il adopte pour père. Le délai de garde permet au père de laisser le temps à un éventuel message "rentrer" d'arriver. Un Δ pour que le message arrive au fils potentiel, un Δ pour que sa réponse éventuelle ait le temps d'arriver. Si au bout de 2Δ, le père n'a rien reçu, c'est que le fils potentiel n'est finalement pas un fils.

2. Indiquer ce qui se passe sur le réseau ci-contre dans le cas suivant. Dans une première phase, le leader envoie deux calculs vers x. Le premier passe par le dessus, passe par x puis arrive en y où il s'éteint. Le deuxième passe par le dessous, arrive en x après l'autre mais moins de 2Δ plus tard, puis est propagé vers y, puis vers le cercle de droite dans lequel les calculs durent longtemps, puis le calcul remonte au leader.
   Ceci entraine une deuxième phase similaire à la première,
   mais où les règles du dessus et du dessous sont inversées,
   alors que l'arbre de la première phase ne s'est pas encore rétracté

TD 9 08/04/2026 (ne rentre pas au partiel)

Exo 1 : 4 processus, ressources A, B, C, D (accès en exclusion mutuelle)

wait or die / wait or wound : règle de priorité se font selon le temps global de la 1ère demande

↑ retente son calcul 1 min plus tard

temps au pire : ≈ 2 heures

au mieux : ≈ 1 heure

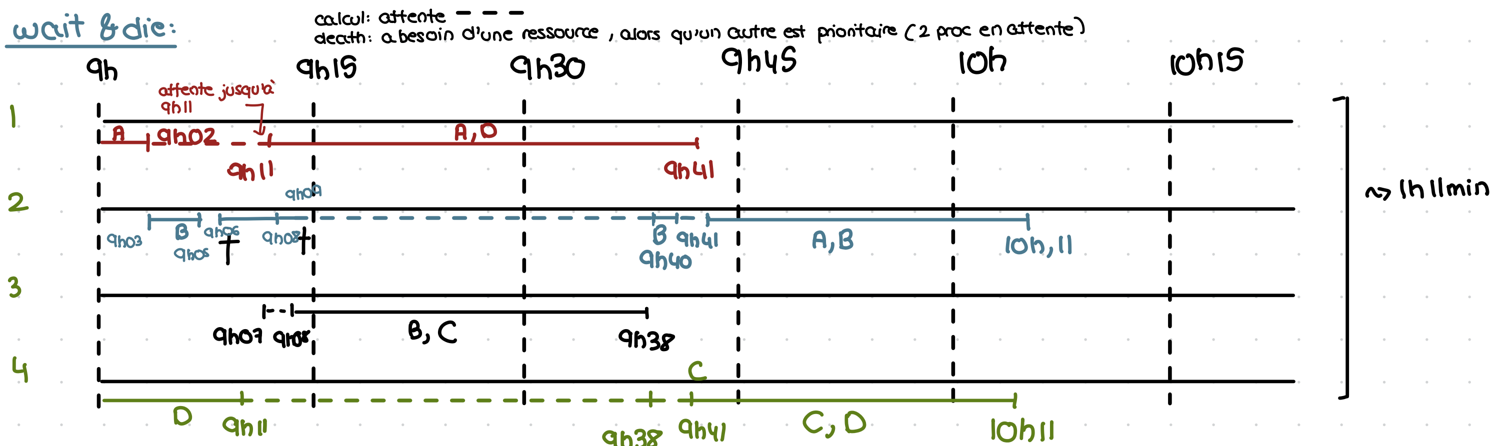
die : perd toutes ses ressources

wound : perd une ressource

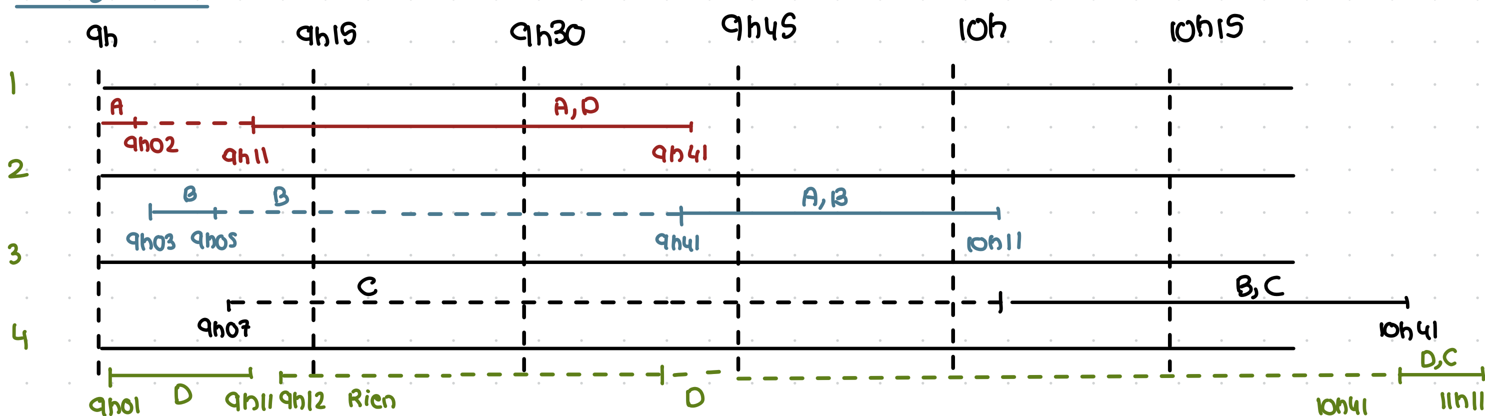
Scenario 1 :

1.
- À 9h00, le processus 1 veut effectuer un calcul qui nécessite la ressource A pendant 2 minutes, puis les ressources A et D pendant 30 minutes.
- À 9h01, le processus 4 veut effectuer un calcul qui nécessite la ressource D pendant 10 minutes.
- À 9h03, le processus 2 veut effectuer un calcul qui nécessite la ressource B pendant 2 minutes, puis les ressources A et B pendant 35 minutes.
- À 9h07, le processus 3 veut effectuer un calcul qui nécessite les ressources B et C et qui durera 30 minutes.
- À 9h12, le processus 4 veut effectuer un calcul qui nécessite les ressources C et D et qui durera 35 minutes.

wait & die :



wait & wound :

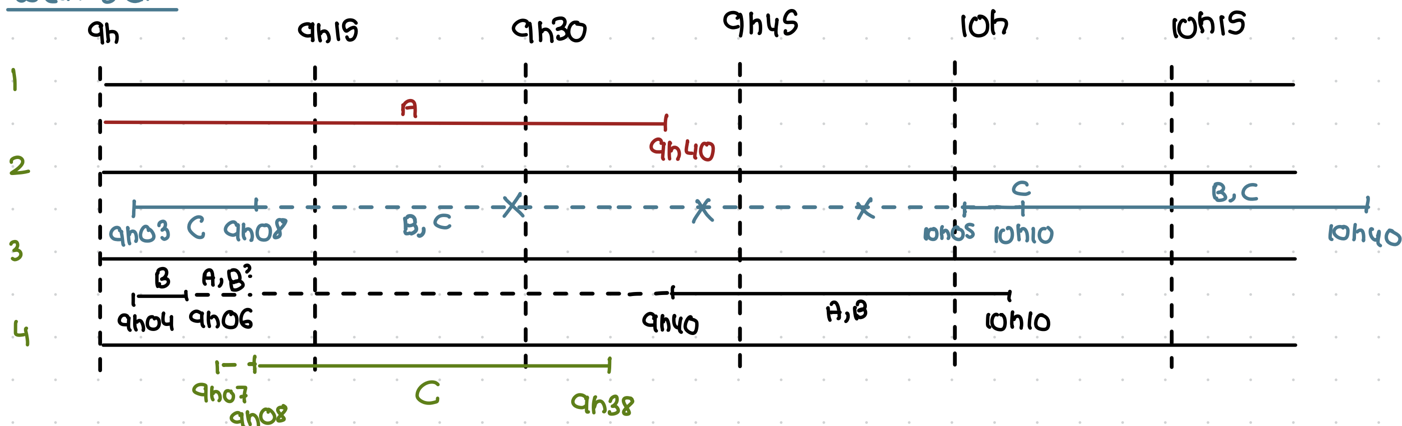


~> 2h11 min

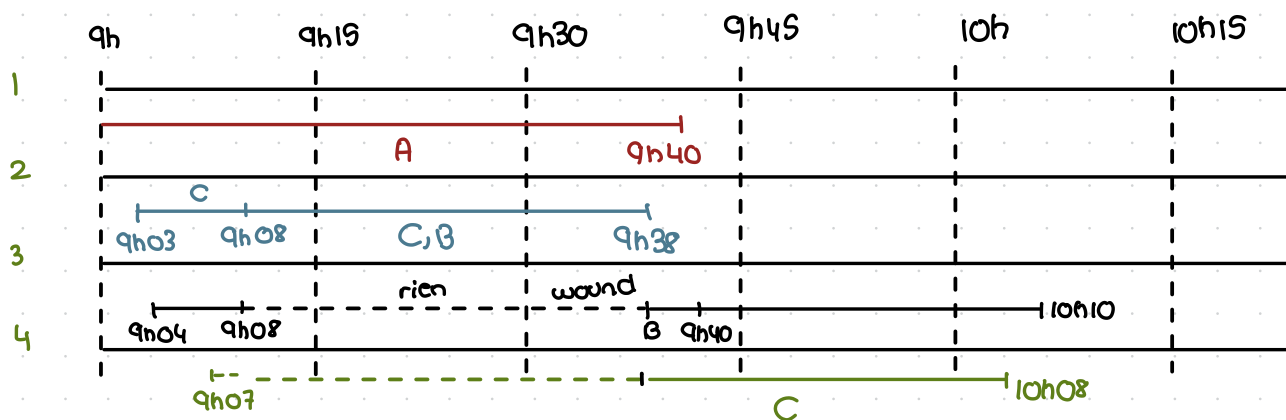
Scenario 2 :

2. • À 9h00, le processus 1 veut effectuer un calcul qui nécessite la ressource A pendant 40 minutes.
   • À 9h03, le processus 2 veut effectuer un calcul qui nécessite la ressource C pendant 5 minutes, puis les ressources B et C pendant 30 minutes.
   • À 9h04, le processus 3 veut effectuer un calcul qui nécessite la ressource B pendant 2 minutes, puis les ressources A et B pendant 35 minutes.
   • À 9h07, le processus 4 veut effectuer un calcul qui nécessite la ressource C pendant 35 minutes.

wait & die :



wait & wound :



Exo 2 :

réseau uniforme, 2 processus , lien bidirectionnel, élection

1. • Mq ∄ algo déterministe anonyme qui réalise l'élection
2. • élection avec proba 1/2
3. • ———————— 1 + complexité en moyenne. $\frac{1}{2^K} \xrightarrow[K \to +\infty]{} 0$

phases en moyenne : 2

taille N + complet :

$$\sum \frac{\log_2 n^2}{2^K} \longrightarrow \frac{4}{3} n^2$$