Notation : 50-50 devoirs (2-3 faisables en binômes) + examen (documents papier autorisés)

Cours n°1 - 02/01/2017

Pas de notion d'état global, chaque système a son propre état local.

Système distribué ⇒ ensemble de processus. Un processus est une entitée identifiée ou anonyme (on la connaît pas ou elle veut pas qu'on la connaisse). On va avoir du mal à distinguer deux processus l'un de l'autre . On va supposer que les processus connaissent cet ensemble Π : on sait avec qui on fait l'algorithme, ce n'est pas comme un système P2P où n'importe qui peut s'ajouter à l'algorithme. On a un ensemble de processus donné et on sait avec qui on est. Les autres n'existent pas.

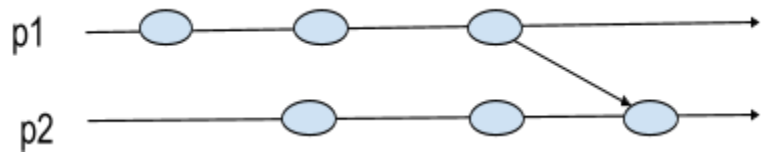
Les processus sont nommés (soit des entiers 1, 2... soit p1, p2...) et on a un nombre fini de processus. Quand on a besoin d'une variable muette, le processus s'appelle p ou q. De manière classique, m également.

Les processus communiquent (par messages pour ce cours). La communication par messages va être une ***communication point à point*** d'un processus p à un processus q (p envoie un message m au processus q et q va recevoir ce message m).

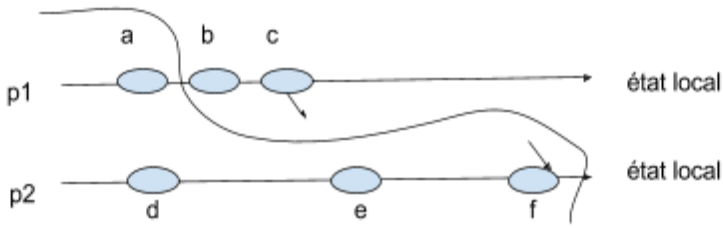
Il y a d'autres types de communication : toujours par messages, une ***communication par diffusion*** (le processus envoie un message qui est diffusé à tous les processus du système, c'est le système des ondes radios), mode de ***communication par mémoire partagée*** (typiquement ce qu'on a quand on a des processus sur une même machine). Ils communiquent via la mémoire.

La communication par message est équivalente à celle par mémoire partagé : celle par mémoire partagée peut être simulée par messages (mais les problèmes ne sont pas les mêmes, les techniques non plus).

On va supposer ici que la communication est ***fiable*** : le processus p qui envoie un message au processus q et un jour le message sera reçu, il ne sera pas dupliqué, il n'est pas modifié, il n'est pas corrompu ; toutefois, il y a les ***délais de réception*** entre l'envoi et la réception.



Synchrone : je sais à partir du moment où j'émet un message, à quel moment il sera reçu (chaque processus à une horloge interne et il sait mesurer le temps). Quand on a un système synchrone, on a pas mal de connaissances sur le processus. On a des connaissances sur l'état global. Le problème, c'est qu'il faut cette connaissance : s'il y a une panne ou si on a mal apprécié le délai, l'algorithme ne va pas vouloir fonctionner. Les algorithmes ici sont fragiles et ne supportent pas les pannes potentielles qui pourraient survenir. Pour essayer de tenir compte de ces pannes, on va essayer de construire un algorithme asynchrone. On ne fait aucune hypothèse sur les temps de transmissions de message et des processus. On suppose simplement qu'une fois envoyé le message est reçu.



Asynchrone : c'est plus compliqué mais également plus robuste. On a pas de correspondance de temps entre les deux processus. Le premier processus ne saura pas quand le message sera reçu par rapport à son temps à lui. Chaque processus a sa propre horloge et est capable de mesurer le temps, mais ces horloges ne sont pas les mêmes d'un processus à l'autre et ne sont pas synchronisées. Chacun a sa connaissance locale de sa propre horloge. Capable de mesurer le temps de leur propre horloge.

Ordre causal $\lessdot$: sur un processus, les événements sont ordonnés : send(m) $\lessdot$ receive(m) pour chaque message. Sur chaque processus, les événements Tg ont lieu à des instants différents.

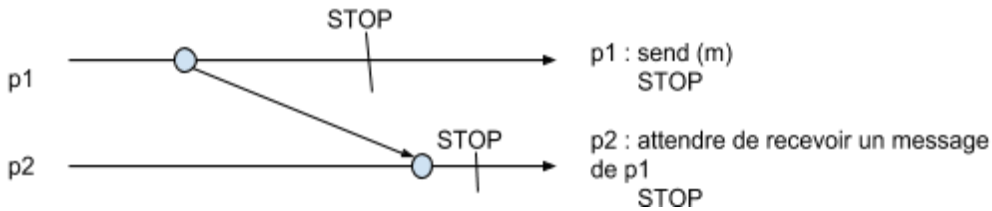
$a \lessdot b \lessdot c$ $\quad$ $a \lessdot c \lessdot f$

$d \lessdot e \lessdot f$ $\quad \Rightarrow \quad$ $a \lessdot f$

$c \lessdot f$

$a \,||\, d \Leftrightarrow$ non $a \lessdot d$ et non $d \lessdot a$

Une exécution est un ordre total : $c1 \xrightarrow{a} c2 \xrightarrow{d} c3$



Quand c'est asynchrone, quand un processus fait stop, on ne sait pas si l'autre processus a fait stop. On a donc une connaissance locale au processus et non globale.

Exemple : Terminaison Distribuée

On a un ensemble d'employés qui ont sur le bureau une pile de dossiers à distribuer. Ils ont un système pour transmettre messages et dossiers. Chaque employé peut soit traiter son dossier ou le transmettre. Etat local : Chaque employé possède sa pile de dossiers. Etat global : Aucun des employés de l'immeuble n'a de travail. Communication ***synchrone*** (téléphone).

Etat local d'un processus : soit il va être actif, il va être passif. Au début de la journée, il est actif et puis il traite ses dossiers, à un moment il n'a plus de dossiers à traiter donc il va passer d'actif à passif. Il reçoit un coup de téléphone : il passe de passif à actif. C'est ce qu'on appelle l'**algorithme de base**, c'est ce que sont en train de faire les processus. L'algorithme de base décrit comment est traité un dossier, que fait chaque processus pour ça, à quel moment il choisit la personne qu'il doit appeler...

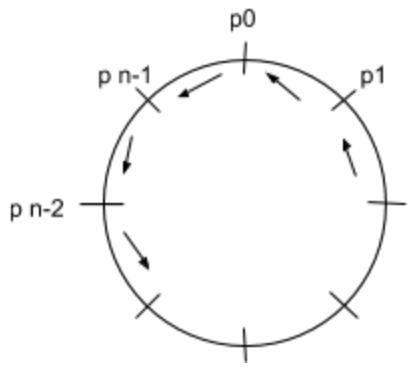
Mais on veut que ces employés puissent rentrer chez eux : on veut détecter la terminaison. La terminaison, c'est *quoi* ? Ca veut dire que tous les processus sont passifs. A partir du moment où cette propriété là s'est produite, elle devient stable et elle ne peut plus être réveillée. Une fois qu'elle est vraie, elle reste vraie pour toujours, plus rien ne peut se passer. Le but, c'est de la détecter : la détecter apporte une information, puisqu'elle reste tout le temps vraie. Détecter la terminaison veut dire que tous les processus doivent avoir la connaissance de la terminaison.

En ***asynchrone*** (principe du coursier qui transmet les messages). On se retrouverait avec un message en transit : on peut avoir tout le monde en passif, puis quelqu'un qui se réveille. Si la communication n'est pas instantanée, on doit rajouter à la terminaison le fait qu'on a aucun message en transit.

On veut superposer à l'algorithme de base un **algorithme de détection de terminaison**. Cet algorithme n'a pas le droit d'intervenir sur les variables de l'algorithme de base. Mais il peut tout à fait regarder l'état du processus et des variables. Pas le droit de geler le système. Si on fait une détection en demandant à tout le monde l'état, le temps qu'on arrive à la fin des interrogations, quelqu'un d'autre peut se réveiller.

- *Solution actuelle* : on fait une première enquête où on demande à le monde s'ils sont actifs ou passifs (si actif, on sait que c'est raté). Si tout le monde est passif, on refait une enquête pour savoir si quelqu'un a changé d'état. Si tout le monde dit "passif et n'ait pas été actif depuis la dernière fois où on m'a demandé", celui qui interroge est au courant, mais pas les autres. Si les autres font l'enquête, celui qui a fini ne répondra pas donc ça ne fonctionnera pas : il faut pouvoir informer les autres.
  Mais ça risque d'être coûteux, de contacter tout le monde : on aimerait pouvoir diminuer ce coût (on envoie n messages, on en reçoit n, on fait la seconde vague, 2n, puis on envoie l'information aux autres, donc n, le tout pour chaque personne...).
  [pas entendu la première manière de faire].
  La seconde manière peut être faire de manière plus symétrique : on va organiser les processus. On ne va pas faire une interrogation au hasard, mais organiser ça de manière à avoir un voisin de droite et un voisin de gauche : on demande à un des voisins, qui mettra l'information sur le message et enverra à ses propres voisins. Le message va transiter entre les voisins. Quand le message aura un RATÉ (personne active). Si mon message revient avec RÉUSSIT (première phase, tout le monde est passif), on recommence pour faire la seconde tentative.

Algorithme de détection de terminaison



arrêt = faux
init : couleur = noir
couleur = blanc
(1) p_0 attends que state = passif
envoyer(détection, OK) à p_{n-1}
attends de recevoir un message de p_1 (détection, -)
soit (détection, a) ce message.
si a == KO alors aller en (1)
si a == OK alors si couleur == blanc && state = passif
alors envoyer (FINI) à p_{n-1}
arrêt = VRAI
sinon aller en (1)

[sur passage de passif à actif couleur = noir]

Tous les processus p_i i != 0

sur réception de (détection, -)

soit (détection, a) ce message

si a == OK && couleur == blanc && state == passif

alors envoyer (détection, OK) à p_{i-1}

sinon attendre state == passif

envoyer (détection, KO) à p_{i-1}

couleur = blanc

sur réception de (FINI)

si i $\neq$ 1 alors envoyer(FINI) à p_{i-1}

Terminaison $\Rightarrow$ $\exists$ un état e de l'exécution tel que $\forall$ p e ($state_p$) = passif

$state_p \leftarrow$ variable locale state du processus p

Détection

Propriété de vivacité (Liveness)

Pour toute exécution e,

si $\exists$ un état e où $\forall$ p $state_p$ = passif

alors $\exists$ un état de e où $\forall$ p arret = vrai.

Terminaison $\Rightarrow$ FG Détection

algo de base — algo de détection

Pour tout execution de e,

s'il existe un état x de e tel que $\forall$ p $arret_p$ = vrai

alors $\forall$ p $state_p$ = passif

sûreté (safety)

Initialement, t = 0

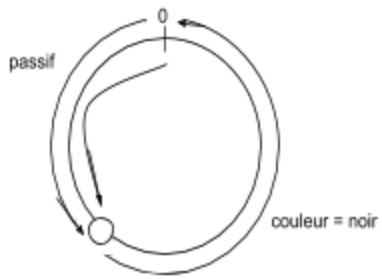
t : le processus qui a reçu le message détecté et qui ne l'a pas encore transmis.

p_0 = $\forall$ p_i (N > i > t) $state_{pi}$ = passif

p_1 = $\exists$ j (t $\geq$ j $\geq$ 0) $couleur_{pi}$ = noir

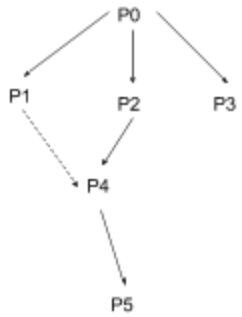
p_2 = le message reçu est (détecté, KO).

Évaluer à la réception du message.



p_0

- passif
- couleur = blanc
- reçu = OK



Etat global :

$\forall$ p $\in \pi$ état local de p

$\forall$ p,q $\in \pi^2$ état du canal de p à q

P_0 = actif

Pi avec i $\neq$ 0 $\rightarrow$ passif

communication asynchrone

init : père $p_0 = p_0$

père $p_i = \perp$

$\Rightarrow$ S'assurer qu'on a rien dans les canaux.

On va supposer que le matin quand on arrive, il n'y a pas de pile de travail sur le bureau. Un processus possède la pile sur son bureau et distribue l'activité.

p0 est le seul processus actif et tous les autres sont passifs.
La communication est asynchrone et c'est p0 qui va distribuer le travail.
On va construire l'arbre de réveil - qui p0 a réveillé.
p0 envoie du travail à p1, p2, p3.
p2 réveille p4 - il pourrait réveiller p3 mais ça ne serait plus un arbre, donc on évite.
p4 réveille p5.
p1 pourrait également réveiller p4, mais pour conserver un arbre, on va éviter.

Si on arrive à faire un arbre de réveil et si chaque processus qui a terminé son activité sort de l'arbre, mais qu'on conserve un arbre en assignant p0, alors on aura gagné parce qu'on aura les réveils. On saura qui a réveillé à qui.
Quand on rentre dans l'arbre, on est positionné quelque part et on sort de l'arbre quand on est passif.

Si p0 à la fin se retrouve seul et passif, c'est gagné.
L'arbre correspond correspond à une donnée globale et toujours le même problème : on a pas de donnée globale, on doit la construire. On va la distribuer : un processus doit savoir qui est son père. Si chacun des processus sait qui est son père, si on récolte ces informations là, alors on a les données de l'arbre. Le maintien de l'arbre au cours de l'exécution de l'algorithme, c'est le maintien de qui est leur père. Donnée globale : vision de l'observateur qui sait tout sur tout. Il suffit, pour le faire, d'avoir la donnée de chacun des processus. Chacun des processus va avoir une variable père qui va lui indiquer qui est son père dans l'arbre de réveil. Avec le père de p0 c'est p0, le père de pi c'est $\perp$ (indéfini, j'initialise la variable par une valeur qui n'est pas prise par une valeur de processus ou que l'algorithme utilise après).

La difficulté est : le premier message de réveil, c'est le père. Si p2 est réveillé par p3, il faut que p3 n'ait qu'un seul père : pas de liaison entre p2 et p3, on ne rajoute pas le lien. Si p4 est réveillé par p2, même si p1 le réveil, il ne reçoit pas. Le message qui fait passer de passif à actif, c'est le père.

La difficulté est de voir qu'on a fini.

Si on a réveillé personne, quand on devient passif, on a fini (donc on s'enlève de l'arbre). Si tous les fils sont enlevés de l'arbre, on a fini. Vraiment ? Oui. Donc on peut s'enlever de l'arbre. Mais il y a le problème de, par exemple, p1 qui a réveillé p4, mais p4 n'est pas vraiment un fils de p1 : pour que p1 ait fini, il faut que ?????. Mais quand p4 finit, il n'envoie un message qu'à p2. Il faut un mécanisme pour savoir que tout ceux qu'on a réveillé ont terminé.

Il faut savoir quand se rajouter, quand s'enlever de cet arbre.

Il faut savoir, si quand on est passif et qu'on veut se retirer de l'arbre (passer le père à un état défini) que tous les gens que l'on a réveillé ne sont pas dans votre descendance dans l'arbre (quelqu'un d'autre les surveille). p2 surveille p4 et p5, si p2 se retire, il faut que p4 et p5 aient terminé. p1 qui a réveillé p4 peut se retirer, parce qu'il sait que p4 est surveillé par p2. L'arbre des réveils, c'est aussi l'arbre de qui surveille qui. L'idée est qu'un processus surveille les processus qui sont en dessous de lui dans l'arbre.

⇒ Trouver un mécanisme pour que cet arbre soit géré correctement.

Idées :

- chaque processus ne réveille qu'un seul processus : **FAUX**, l'algo de base est celui de distribution du travail, on n'a pas le droit de le changer, il doit pouvoir donner du travail à n'importe qui. Pas le droit de toucher à l'algorithme de base, on peut seulement observer les valeurs.
- On essaie d'avoir le moins de messages possible échangés. On veut le plus simple possible (les preuves en distribué sont difficiles).

Solution :

- On va compter les messages qu'on a envoyé.
- Quand quelqu'un est réveillé par un autre, il le prend pour père, donc il sait qu'il est surveillé. Et les autres qui lui font la demande, il lui répond "j'ai bien reçu le message et je suis surveillé".
- Si on incrémente la variable ***compte*** à chaque fois qu'on envoie un message, on décrémente à chaque fois qu'on reçoit un message disant que l'autre est surveillé (il a un père).
- Si cette variable ***compte*** est à 0, ça veut dire que tous ceux que j'ai réveillé ont répondu qu'ils étaient surveillés par quelqu'un d'autre => je n'ai pas d'enfant sous moi.
- Quand j'ai un père (j'ai été réveillé par quelqu'un), à tous ceux que je réveille j'envoie un message, mon père lui aura son compteur positif (je ne lui ai pas répondu), je vais lui envoyer une réponse "j'ai fini et tout ceux que je surveille ont fini". Mon compteur qui est passé à 0 est que tous ceux que j'ai contacté sont soit surveillés par un autre, soit ont fini.

<u>Cours n°2 - 09/01/2017 :</u>

Algorithmes de vague

Problème :

- Diffusion d'information, on veut qu'une information soit disponible dans tout le réseau
- On doit gérer la synchronisation. On veut que tous les processus aient fait une certaine instruction avant de passer à la suivante. On veut récupérer l'information via cet algorithme.

On suppose qu'on a une exécution C, (avec n processus, une communication point à point asynchrone et un graphe connexe : tout le monde peut communiquer avec tout le monde), qui est une suite d'événements partiellement ordonnée par la relation (séquentielle). Les processus envoient un message aux autres processus, et l'émission est avant la réception, c'est ce qui donne l'ordre. On note | C | = le nombre d'éléments (#) de C. De plus, on a un algorithme, avec un élément distingué qu'on appelle **décide** (?)

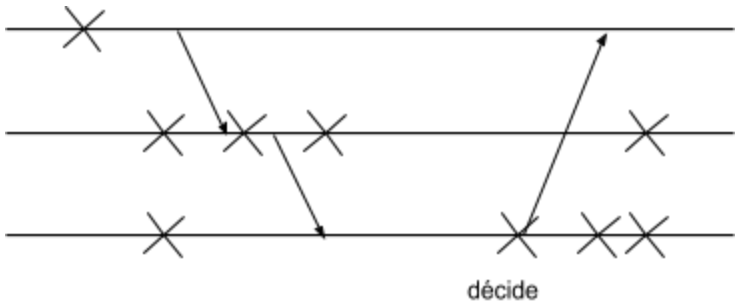
Définition : Un algorithme est un **algorithme par vague** s'il satisfait les conditions suivantes :

- ***Terminaison*** : plus aucun processus n'exécute d'événement (au bout d'un moment, il ne se passe plus rien)

  $\forall$ C | C | est fini, $|C| < \infty$
- ***Décision*** : on veut que pour toute exécution il y ait un évènement e tel que e = décide p

  $\forall$ C $\exists$ e ϵ C $\exists$ p tel que e = decide p
- ***Dépendance*** : quand l'événement décide s'est produit, il y a eu un évènement sur chacun des autres processus.

  $\forall$ C $\forall$ e $\in$ C (événement décide p $\Rightarrow$ $\forall$ q $\in$ π $\exists$ f $\in$ Cq f $\leqslant$ e)



Cet algorithme de vague, il faut le déclencher. Il y a des **initiateurs.** L'initiateur débute l'algorithme de manière spontanée, et les **suiveurs** (non initiateurs) débutent l'algorithme de vague sur réception d'un message.

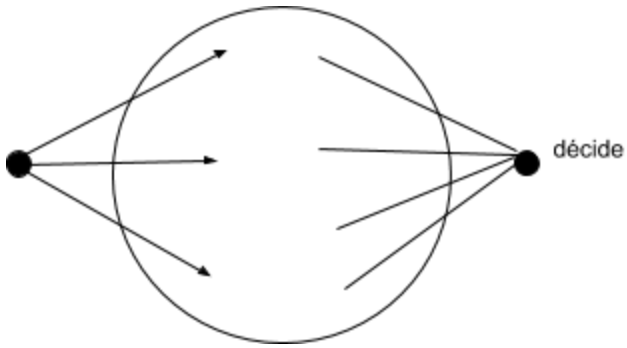
Différents algorithmes de vague :

- **Centralisé** : ils ont un seul et unique initiateur, un seul processus déclenche l'algorithme (sinon, ils sont nommés décentralisés)
- **Topologie du réseau** : Anneau, arbre... On regarde si les processus ont déjà une structure ou si c'est l'algorithme qui va la leur donner.
- **Nombre de décision** : Est ce qu'on en a une seule, une ou plus, est ce que tous décident? et la nature du décideur.

Dans ces algorithmes, il y a une sous-classe qui joue un rôle important, ce sont les **algorithmes de traversées** :

- Un seul initiateur
- Un seul décideur qui est l'initiateur

Typiquement, ces algorithmes sont ceux pour lesquels un processus veut diffuser un message et le diffuse aux autres processus. Comme il réveille tout le monde, on considère que l'information est parvenue à tout le monde. Ce genre d'algorithme a beaucoup été étudié.



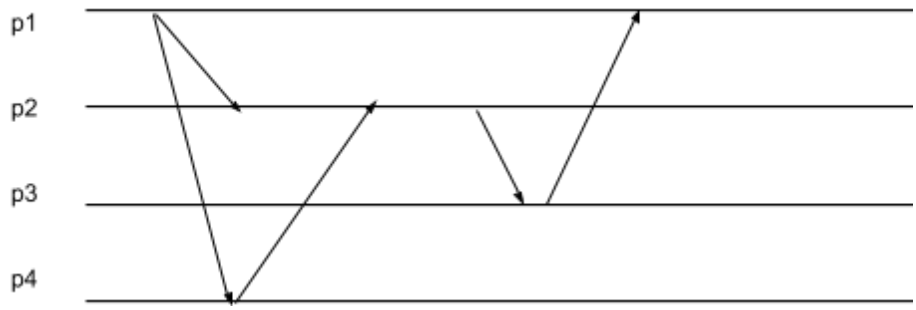
Quelques propriétés :

- **Propriété 1** : Une exécution d'un algorithme de vague avec un seul initiateur q.

  ∀ q != p, père q est *l'émetteur du premier message* que reçoit q

  ⇒ *Est ce que c'est bien défini ? Est ce que j'ai toujours l'émetteur du 1er message que reçoit q ? Est que tous les processus reçoivent un message ?*

  $T = (\pi, \{(q,r)\ ;\ q\ !=p\ \wedge r\ =\ père\ q\}$ est un arbre de racine p



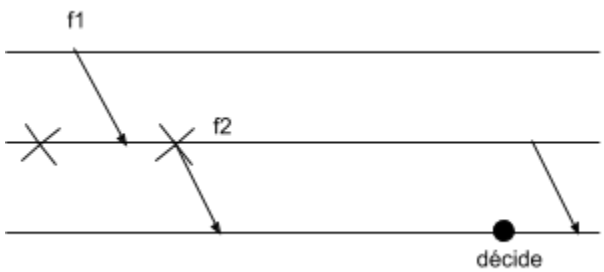
G un graphe de n sommets

- G connexe
- G a n-1 arêtes
- G est sans cycle

⇒ alors G est un arbre.

- **Propriété 2** : Si C est une exécution d'un algorithme de vague et décide p une décision sur p alors :

  ∀ q != p ∃ f ∈ Cq f ⩽ dp et f est une émission.



- **Propriété 3** : Si C est une exécution d'un algo de vagues avec un unique initiateur qui est un décideur ⇒ au moins | π | messages échangés.

Algorithme sur un anneau

- Centralisé

Soit p0 l'initiateur (algorithme de traversée)

Code de p_0 :

```
envoyer < vague > à p1
attendre de recevoir < vague > de pn-1
décide
```

Code p_i != p_0

```
sur réception de < vague > de pi-1
envoyer < vague > à pi+1
attendre < fini > de  pi-1
decipe p i
si  pi !=  pn-1
       envoyer < fini > à  pi+1
```

(1) pour que tous les processus décident :
on fait un second tour ⇒ 2n - 1 messages

Algorithme de vague sur un arbre

Initiateurs : feuilles
Décideurs : un ou deux processus

Code de p :

```
var     rec[q] boolean pour tout q voisin de p
               init false
begin   while # { q rec[q] = false } > 1
        do
               receive < vague from some q
               rec[q] = true
// il n'y a plus qu'un seul q0 tel que rec[q0] = false
Soit q0 tel que rec[q0] = false
send < vague > to q0
receive < vague > from q0
rec[q0] = true
decide

∀ q voisin de p        q != q0        do send < vague > to q
```

Combien de messages sont envoyés par chaque processus ? 1 au plus.

Théorie/Théorème/Aucune idée : L'algorithme d'arbre est un algorithme de vague.

- *Terminaison* : comme chaque processus envoie au plus un message avant le "decide", après N messages au plus l'algorithme atteindra une configuration terminale.
- *Décision* :

Soit δ une configuration terminale (aucun message en transit - tous les processus ont fini ou sont bloqués sur une réception).

{ $rec_p[q]$ = faux dans δ | p $\in \Pi$, q $\in$ voisin de p }

K = # de processus qui ont envoyé un message avant δ

Code de p :

F = #{ rec_p [q] = faux dans la configuration initiale } = 2 (N - 1)

F = 2 (N - 1) - K >= 2 (N-K) + K

2N-K-2 >= 2N - K IMPOSSIBLE

On suppose qu'il n'y a pas d décision.

N - K processus qui n'ont pas envoyé de message.

soit x un tel processus # { $rec_x[q]$ = faux } >= 2

K processus qui ont envoyé un message

soit x un tel processus rec_x $[(q_0)_x]$ = faux

F <= K + 2(N-K)

===> DECISION

Dépendance

f_{pq} émission de p → q

g_{pq} réception de p → q

<u>Cours n°3 - 16/01/2017</u>

Algorithmes de vague : 3 propriétés

- ***Terminaison*** : l'exécution se termine. On ne peut plus évoluer.
- ***Décision*** : il doit y a un événement spécial, c'est l'événement decide.
- ***Causalité*** : quelque soit l'événement decide (il en faut au moins un, mais on peut en avoir plusieurs), pour n'importe quel processus il y a un événement ??? il doit y avoir des nouvelles (directes ou indirectes) pour prendre une décision.

Algorithme sur un arbre

Initiateurs : feuilles

On sait qu'on a une feuille parce qu'on a un seul voisin.

Comment fonctionne l'algorithme :

- Les feuilles envoient un message.
- Les pères des feuilles reçoivent un message de toutes leurs filles (de tous leurs voisins), sauf d'une (puisqu'ils vont envoyer un message à ce voisin là).
- Quand on a reçu un message de tous ses voisins, on décide.

Code :

```
var rec_p[q]    q ϵvoisins de p       init false
while # { q : recp[q] est faux } > 1
        do {
                receive < tok > from q
                rec_p[q] = True
        }
/* soit q_0 et q rec_p[q_0] = faux */
        send < tok > to q_0
        receive < tok > from q_0
        rec_p[q] = true
        decide
```

ϑ une configuration terminale.

F = # { $rec_p[q]$ = Faux dans ϑ | ∀ p ∈ ϖ

⬇ ∀ q ∈ voisin de p }

⬇

pas de messages de p ⟶ q

K = # de processus qui ont envoyé un message

Il n'y a pas de message en transit.

Que vaut F ?

F = 2 (N-1) - K s'il n'y a pas de décision (aucun processus n'arrive à decide)

F ≥ 2 (N-K) + K

~~2N~~ - 2 - ~~K~~ ≥ ~~2N~~ - ~~K~~ ⇒ **CONTRADICTION**

L'hypothèse qu'il n'y a pas de décision est donc FAUSSE : il y a une décision.

Montrer la dépendance :

Considérons qu'il y a un lien de p à q.

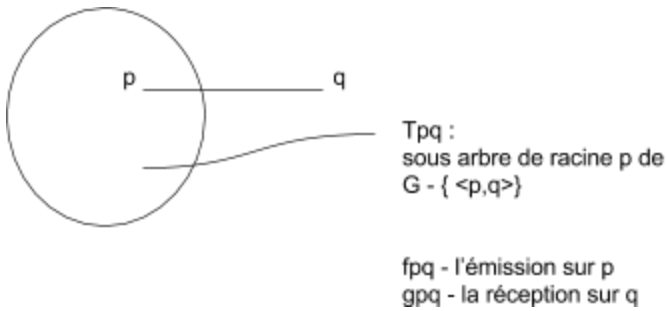
∀ s ∈ Tpq ∃ e ∈ ϑ e ≤ gpq

Par induction sur l'ordre des émissions dans ϑ.

Pour la première émission :

Tpq = { p }

sur p : fpq ≤ gpq



À la r+1e réception

c'est une réception sur q p ⟶ q

⇒ p a reçu un message de tous ses voisins sauf q avant

Pour un voisin r

r ⟶ p un message de r ⟶ p

∀ s ∈ Trp ∃ e ∈ ϑs

$e \leq grp$

$Tpq = \{ p \} \cup Trp$ $\quad r \in voisin\ p - \{q\}$

$\forall\ s \in Trp$ $\quad s = p$ $\quad fpq \leq gpq$

$s \in Trp$ $\quad \exists\ e \in \delta s$ $\quad e \leq grp \leq fpq$

$fpq \leq gpq$

$\Rightarrow e \leq gpq$

[X_X] <('.'<)

Soit p un processus qui décide.
Quand il décide, alors il a reçu un message de tous ses voisins

$\Rightarrow \forall\ x \in$ voisin de p, un message $x \rightarrow p$

$\Rightarrow \forall\ s \in Txp$ alors $\exists\ e \in \delta s$ tel que $e \leq gxp \leq$ decide

$\Rightarrow \forall\ s \in \pi \quad \exists\ e \in \delta s$ tel que $e \leq$ decide

G graphe connexe (non orienté), centralisé
Algo de probe/echo

var rec : int initialisée à 0
var pere : processus (π) , initialisée à $\perp$

Initiateur :

```
for all q ∈ Voisin_p do send (< tok >) to q
while rec <  #Voisin_p do
        receive (< tok >) ; rec++
decide
```

Code d'un non initiateur p :

```
debut
        receive < tok > from q ; pere = q ; rec++
        for all q ∈ Voisin_p and q != pere
                send (<tok>) to q
        while rec < #Voisin_p do
                receive <tok> ; rec++
        send <tok> to pere_p
end
```

On va utiliser les arêtes de l'arbre couvrant pour faire transiter les messages.

Pour le routage, on doit savoir à qui s'adresser, pour ça on a un tableau

R : tableau $\forall\ i \in \pi$

R[i] : le nom de processus à qui envoyer le message pour i indice $\perp$

recevoir <Tok,N> de j

alors $\forall\ x \in N$ R[x] = j

send <Tok,N> à son père

avec $N = \{x / R[x] != \perp\} \cup \{p\}$

Un processus p émet au plus |Voisin p| messages.
Soit δ une exécution, il existe un instant où plus aucun évènement ne peut se produire.

ϑ une configuration terminale

Proposition : Tous les processus p != p_0 ont $père_p$!= ⊥

Démonstration : Par récurrence sur la distance à p_0

T = { π } <p,$père_p$>, p ∈ π {p0} >

⬇

E

O = les arêtes du graphe initial

Tp = le sous arbre de racine p dans T

Soit fp l'événement (s'il existe) p envoie un message à $père_p$,

gp la réception (si elle existe) de cet évènement

1) ϑ contient un évènement fq pour p!=p_0

2) ∀ s ∈ Tp ∃ e ∈ Cs tel que e <= gp

On montre que HR(1) est vraie :

Est ce que c'est vrai pour une feuille?

Comme tous les processus sont dans T ⇒

ils ont envoyé un message à tous leurs voisins sauf peut-être leur père

(HR1) p est une feuille

⇒ p a envoyé un message à tous ses voisins

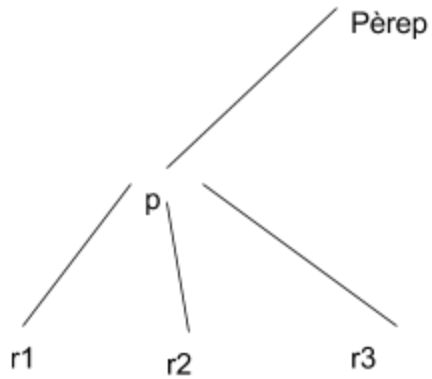
p va envoyer un message à son père

⇒ ∃ fp ∈ ϑ

(2) gp < fp

l'émission du message

HR(k) ⇒ HR(k+1)



En p_0

p ∈ $Voisin_{p0}$-$Fils_{p0}$

p a envoyé un message à p_0 car perep != ⊥

p ∈ Fils p ⇒ p envoie un message à p0

⇒ P0 a reçu un message de tous ces voisins ⇒ decide

Causalité

x ∈ π

$\exists$ v $\in$ $Fils_{p0}$
x $\in$ π_v
$\exists$ e $\in$ γ_x tq x $\leq$ g_r $\leq$ decide

Propagation d'information avec retour : un ensemble de processus connaissent une information (M). On veut qu'un des processus sache que tous les processus connaissent **M.**
Comment utiliser un algorithme de vague pour faire ça? On suppose que les initiateurs connaissent M.

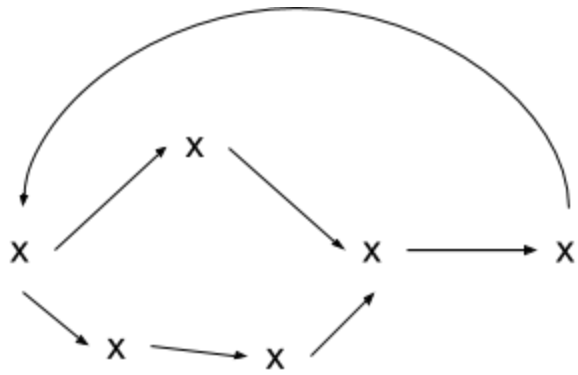
envoyer message <Tok> ⇒ envoyer <Tok,M>

Calcul d'une fonction f à n variables :
le processus pr à la valeur de la var i
But : Calculer la valeur de la fonction avec un algo de vagues
On suppose que les initiateurs connaissent M.

envoyer message <Tok> ⇒ envoyer <Tok,V>

Au moment du décide, on connait toutes les valeurs et ainsi calculer la valeur de la fonction

Cours n°4 - 23/01/2016



Algo de vagues :

- graphe quelconque orienté et connexe

On va essayer d'être le plus général possible pour les initiateurs :

- on en a au moins 1

On veut faire un algo de vagues, et si on ne sait rien de rien (aucune hypothèses) ça va être compliqué voir impossible

Quand l'initiateur reçoit un message de tous ses voisins est ce qu'on a toutes les propriétés de l'algo de base?

- Décision? :

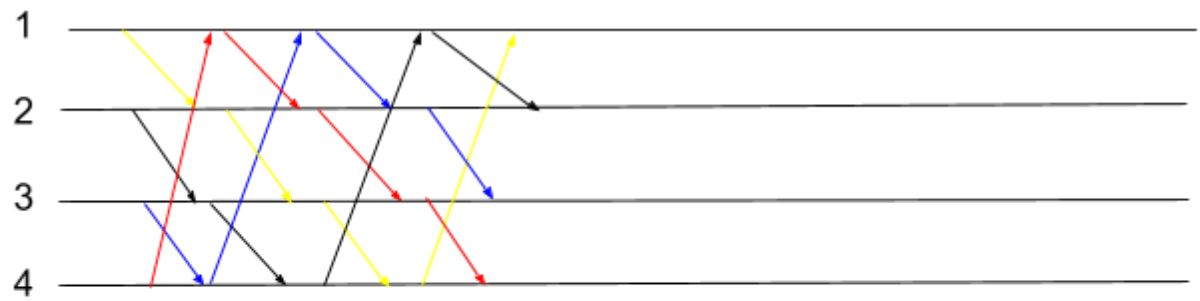
Qu'est ce qu'on peut faire pour s'assurer qu'on a bien reçu un message de tout le monde? :

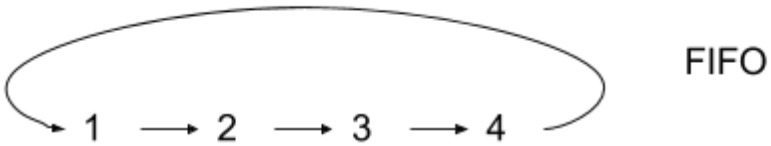
p processus
In p voisins entrants
Out p voisins sortants

D diamètre du graphe
diamètre = max {d(p,q)/ p,q $\in$ π } (D peut être >= au diamètre)

Code de p :

```
Rec [q]  init 0                                  # de messages reçu de q (q ∈ In p)
Sent     init 0                                  # de messages envoyé à Outp
begin if p est initiateur then
        | begin for all r ∈ Outp do { send(<TOKEN>) to r }
      A |       sent ++
        | end
while min Req[q] < D
begin receive(<TOKEN>) from some q
        // soit q0 celui dont on a reçu
        req[q0]++
        if min Req[q] >= Sentp && Sentp < D then
                A
end
```





FIFO

$\forall$ q $\in$ In p $Req_p[q] \geq 3$

min $Req_p[q] = D$

<u>Terminaison</u> chaque processus émet au plus D messages

<u>Causalité</u> $\Rightarrow$ configuration terminale

<u>Décision</u> le processus qui a la plus petite valeur de Sent (> 0)

min $Rec_p[q] \geq Sent_p$

On considère le processus qui a la plus petite valeur de Sent

Dans une configuration terminale

$\forall$ q $\in$ In p $Req_p[q] = Sent_q$

$Rec_p[q] > 0 \Rightarrow Sent_p > 0$

1) p est initiateur OK
2) p non initiateur min $Rec_p[q] \geq Sent_p$ && Sent < D
   $\Rightarrow$ p émet vers ses voisins
   $\Rightarrow Sent_p > 0$

Si $Sent_p < D$ FAUX

$Sent_p \geq D$ $Sent_p = D$
$\Rightarrow \forall$ q $Sent_q = D$
$Rec_p[q] = Sent_q \Rightarrow \min Rec_p[q] = D$

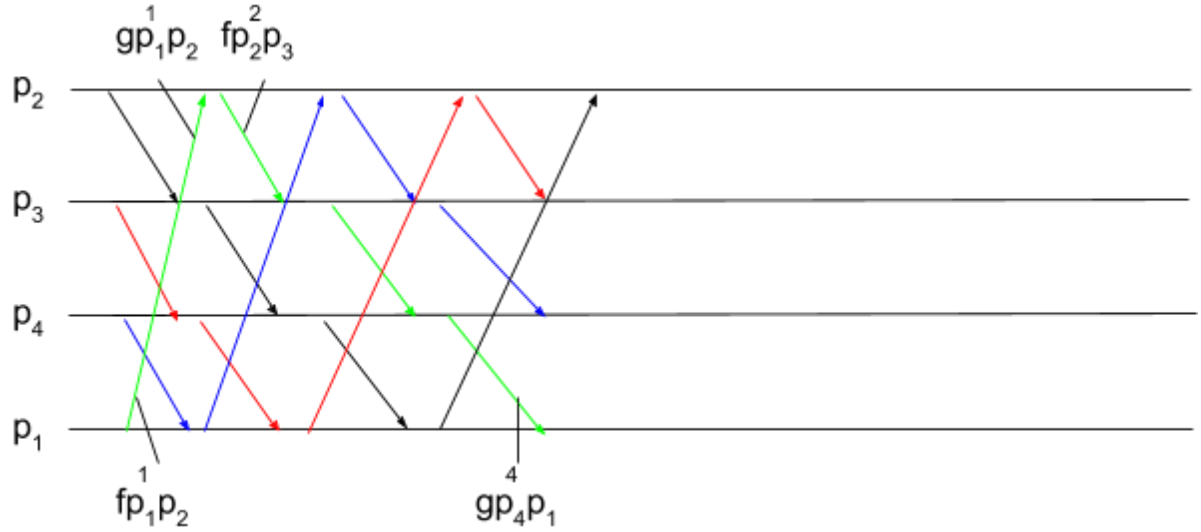
	i	i-ème message entre p et q
p → q	fp_q	émission
	i	
	gp_q	réception
	tous initiateurs + FIFO	

Pour un chemin

$p_0\ p_1 \ldots p_0$ avec $l < D$

(1) (1) (2) (2) (l)

$fp_0p_1 \rightarrow gp_0p_1 \rightarrow fp_1p_2 \rightarrow gp_2p_3 \ldots \rightarrow gp_{l-1}p_l$



nombre de messages = |E| x |D|

Synchronisation :
Un processus p exécute a_p, b_p ,c_p tel que :

- il exécute b_p si tous les processus ont exécuté a_p
- il exécute c_p si tous les processus ont exécuté b_p

Exécute a : algo de vague (tous initiateurs et décideurs)
quand p de code exécute b
Algo de vague jusqu'à décision
exécute c

Algorithme d'élection

Etat initial : Tous les processus dans un même état
But : atteindre un état ou exactement un processus est dans un état leader et les autres sont dans un état perdu.
A quoi ça sert ? Avoir un chef c'est bien, ça sert à trouver un initiateur. Avoir un processus qui répartit les tâches c'est bien.

Un algo d'élection est un algo qui satisfait les propriétés suivantes :

1) Chaque processus à le même algo local
2) L'algo est décentralisé (tout ensemble non vide de processus peut initier l'algo)
3) L'algo atteint une configuration terminale ou un processus est dans l'état leader et les autres dans l'état perdu

Communication asynchrone

- tous les processus ont une identité $\subseteq \pi$
+ une relation d'ordre, graphe connexe

Élection et l'algo de vague

arbre

```
if etat = decide then
        send(dec, p) à tous
        attendre(dec, _) d'un des voisins soit (dec, x)
        if x < p then etat = perdu
        else etat = leader
if etat = leader then
        send(leader, p) aux voisins
```

```
upon receive(leader, q)
if p != q then
        etat = perdu
        send(leader, q) aux voisins
où on a un ensemble L
L = {p}
```

- send(<TOKEN>) ⇒ send(<TOKEN>, L)
- receive(<TOKEN>, L_1) : L = L_1

```
if p decide then
        lead = min L
        if p = lead then
                etat = leader
        else
                etat = perdu
        send(leader, lead) aux voisin p à émetteur du dernier message reçu
```

Cours n°5 - 30/01/2017

Correction du devoir

Dans l'hypothèse où il n'y a qu'un seul initiateur et que les processus sont soit actifs soit passifs. Passage d'actif à passif sur réception d'un message. On passe d'actif à passif spontanément. On ne peut pas modifier ça - on peut juste regarder ce qu'il se passe. ***Ce n'est pas la détection qui fait passer à passif.*** L'algorithme de détection va également avoir un initiateur p_0 : initialement, père = p_0. Initialement, l'état pour p_0 est actif et pour tous les autres il est passif. On va ***observer*** l'algorithme de base. Si dans l'algorithme de base A_p :

- si p envoie un message alors cmp++
- si p reçoit un message de q alors
  - si le père = undef alors pere = q
  - sinon envoie <SIG> à q
- si p reçoit un message <SIG> alors cpt--
- si cmp = 0 && etat = passif && pere ≠ undef alors
  - si p = p_0 alors ANNONCE DE LA TERMINAISON
  - sinon envoie <SIG> à pere, pere = undef

Algorithme de détection :

var cpt ⟶ 0

pere ⟶ undef

Si l'algorithme de base envoie *a* message, alors l'algorithme de terminaison, écrit sous cette forme, enverra autant de message que l'algorithme de base, soit *a*.

Montrer que si les processus sont passifs et qu'on a pas de message en transit, on a la terminaison.

Si on considère le graphe G : <X,V>
Avec X = { p ∈ π / $pere_p$ ≠ undef }
V = (p, $pere_p$) p ∈ X - {p_0}

G est un arbre.
Dans G { q | $pere_q$ = p } = cpt_p

Si l'algorithme détecte la terminaison, on est dans l'état compteur est à 0 et l'état est passif. Le compteur de p_0 est à 0 veut dire que p_0 n'a plus aucun fils, aucun processus n'a de père : le père est à undef, soit ils n'ont jamais eu de père, soit le compteur est à 0 et ils sont à l'état passif. S'il y avait encore un message en transit, il y aurait encore un compteur différent de 0.

Algorithme d'élection
(http://dpt-info.u-strasbg.fr/~stella/enseignement/algos_distr/transp_chapitre4.pdf)
L'algorithme est décentralisé, chaque processus peut être initiateur et procéder à l'élection. On a vu un algorithme avec un arbre, et maintenant on va voir l'algorithme sur un anneau. Chaque processus a un état qui peut être **sleep**, **candidat**, **leader** et **lost**. Tous les initiateurs peuvent potentiellement être élu et les autres peuvent simplement participer à l'algorithme. L'algorithme est le suivant :

```
var     List    liste de processus init { p }
        state = sleep

begin   if p est initiateur then
                state = candidat
                send < TOK, p > to Next_p
                receive < TOK, q > from Some_q
                while p ≠ q :
                        List = List ∪{ q }
                        send < TOK, q > to Next_p
                        receive < TOK, q > from Some_q
                if p = max(L) then state = leader else state = lost
        else
                while true :
                        receive < TOK, q > from Some_q
                        send < TOK, q >
                        if state = sleep then state = lost
```

Chaque initiateur va provoquer *n* messages
Nombre de messages : si k initiateurs, kN messages de taille longueur de l'identité + 1
⇒ $O(n^2)$

Amélioration

- Ne pas envoyer le message qui provient d'un processus avec une identité plus petite.

Tous initiateur : N (nombre de message < TOK ?? >) + N-1 = 2N - 1

Anneau bidirectionnel

k processus candidat
1 essai (2N) = Tous les processus envoient 2 messages
k/2 candidats
\+ 1 moyen pour tester que 1 seul candidat restant.

Anneau unidirectionnel en O(n log n) messages.
(détection des minimums)

Algorithme de Peterson

L'algorithme fait une élection de leader sur un anneau unidirectionnel en O(n log n) messages au plus.

```
var     ci      une identité                    init p
        acn     une identité                    init undef
        win     une identité                    init undef
        state_p (candidat, sleep, leader, lost)     init sleep

begin   if p est initiateur then state = candidat
        while win = undef :
                if state = candidat then
                        send < one, ci >
                        receive < one, q > acn = q
                        if (acn == ci) then /* acn est le min */
                                send < fini, acn >
                                receive < fini, acn >
                         else /* acn est le voisin de droite */
                                send < two, acn >
                                receive < two, q >
                                if (acn < ci) and (acn < q) then ci = acn else state = lost
                else /* si on est pas candidat */
                        receive < m >
                        if m = < FINI, q > then
                                win = q
                                state = lost
                        send < m >
        if p = win then state = leader else state = lost
```

Élection sur un réseau arbitraire (lien bidirectionnel)

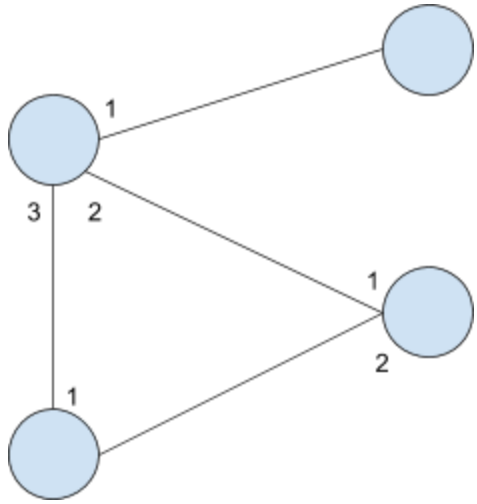
```
var     cw       id d'un processus          init undef
        rcp     integer                             init 0
        father  id d'un processus          init undef
        lrc     integer                             init 0
                win     id d'un processus           init undef
        state   (sleep, candidat, lost, leader)     init sleep

begin   if p est initiateur then
                cw = p
                state = candidat
                for all q ∈ Voisin do send < tok, p > to q
        while lrec < #Voisin :
                receive msg from q
                if msg = < ldr, r > then
                        if lrec = 0 then for all q ∈ Voisin do send < ldr, r > to q
                        lrec = lrec +1, win = r
                else /* soit < tok, p > le message msg */
                        if r = cw then
                                rcp = rcp + 1
                                if rcp = #Voisin then
                                        if cw = p then for all s ∈Voisin send < ldr, p> to s
```

```
                        else send < tok, cw > to father
                if (r < cw) then
                        cw = r ; rcp = 0 ; father = q
                        for all s ∈Voisin s != q do send < tok, r > to s
        if win = p then state = leader else state = lost
```

Cours n°Aucune idée - Je/sais/pas

Réseaux anonymes

Un processus n'a ***pas d'identité***, mais chaque processus à des voisins et est capable d'identifier l'endroit d'où il reçoit des messages de ses voisins. Il n'a aucune raison qu'il y ait coïncidence des numéros de ports entre voisins (par exemple, le port 1 du processus p peut communiquer avec le port 2 du processus q). Ils peuvent ***distinguer leurs voisins***. Comme ils peuvent les distinguer, ils ont ***tous le même code*** pour éviter le fait que le code ne donne une identité.



On va regarder ce qu'il possible de faire dans ce cas avec des ***algorithmes déterministes*** (on a un état, on a une action et on passe dans un autre état). Les algorithmes qu'on a manipulé jusqu'à présent peuvent avoir une dose de non-déterministe qui peut être dû au temps de propagation des messages. Ce non-déterministe n'est pas du code du processus lui-même, c'est du non-déterminisme dans la propagation.

Schéma 2

L'***algorithme probabiliste*** : chaque processus a un générateur de nombre aléatoire. Ce générateur doit avoir de bonnes propriétés : équiprobable (ne pas toujours tiré que des 0 ou que des 1), qu'il ne tire pas la même séquence uniforme (010101), on doit pouvoir tirer n'importe quelle séquence de 0 et de 1). L'hypothèse que l'on fait est que $\forall$ k > 0, toute séquence de k bits a la même probabilité d'apparition dans une suite de tirage .1aléatoire.

Le générateur de nombre aléatoire peut servir pour deux choses :

- soit pour obtenir la terminaison du programme : ***Monte Carlo***, on a toujours terminaison mais on a pas toujours correction (correction probabiliste).
- soit pour obtenir la correction du programme : ***Las Vegas***, on est sûr d'avoir une correction mais pas la terminaison (terminaison probabiliste).

Paramètres :

- La taille du réseau est-elle connue ou pas ?
- **Déterministe** ou probabiliste ?
  - Dans le cas d'un probabiliste, Monte Carlo ou Las Vegas ?

⇒ Déterministe est le plus fort.

- Terminaison ?
  - Arrêt des processus : les processus s'arrêtent.
  - Arrêt des messages : il n'y a plus aucun message qui circulent, les processus sont en attente et ne font plus rien.

⇒ Arrêt des processus plus fort.

Algorithme centralisé / décentralisé

Un seul processus initie l'algorithme : on a déjà potentiellement un leader.

Théorème *:* Il existe un algorithme déterministe centralisé qui calcule la taille du réseau en échangeant des messages, le nombre sera précisé quand l'algorithme sera exécuté.
Rappel *:* un algorithme centralisé possède un initiateur (les autres processus sont non-initiateurs).

G = < v, E >
E bidirectionnel

Algorithme d'Écho : (schéma moche à faire, soso l'a probablement)

```
var :   rec          int     init 0
        pere         port    init undef
        taille       int     init 1
```

Initiateur :

```
begin :for all q ∈ Voisin do send < Tok, 0 > to q
               while rec < #Voisin do
               begin :receive <tok,s>
                              rec = rec + 1
                              taille = taille + s
               end
               /* réseau de taille taille */
```

Non initiateur :

```
begin :receive <tok,s> from port q
               pere = q
               rec ++
               for all r ∈ Voisin r != pere do send <tok,0>  to r
               while rec < #Voisin do
                      begin :receive <tok,s>
                                    rec ++
                                    taille = taille +s
                      end
                      send <tok,taille> to pere
```

```
end
```

Théorème : Il existe un algorithme déterministe centralisé qui nomme les processus. (Schéma moche encore)

Nombre total de messages : 2 |E| - 1

```
var :   rec            int                    init 0
        pere           port                   init undef
        taille int                    init 1
        fils           tableau de int init 0
                       de |Port| élements
```

L'initiateur :

```
begin :         for all q ∈ Voisin do send <Tok,0> to q
                while rec < #Voisin do
                        begin   receive <tok,s>
                                rec = rec + 1
                                t[q]= s
                                taille = taille +s
                        end
                id = taille
                for j = 1 to |Voisin|
                        if T[j] != 0 then
                                send <Nom,i,i+T[j]-1> to j
                                i = i + T[j];
```

Non initiateur :

```
begin :receive <tok,s> from port q
                pere = q
                rec ++
                for all r ∈ Voisin r != pere do send <tok,0>  to r
                while rec < #Voisin do
                        begin :receive <tok,s>
                                        rec ++
                                        taille = taille +s
                        end
                send <tok,taille> to pere
                receive <Nom,j,k> from q
                id = k
                i = j
                for j = 1 to |Voisin|
                if j != q and T[j] != 0 then
                        send <Nom, i,i+T[j] - 1 > to j
                        i = i + T[j]
```

Théorème : Il n'existe pas d'algorithme déterministe pour réaliser une élection dans un réseau anonyme dont la taille est inconnue.

Preuve par contradiction : il existe un algorithme déterministe A qui pour un réseau élit un leader.

On considère un anneau de taille k.

On considère un anneau de taille 2k.

Les processus exécutent A.

On considère une exécution synchrone sur l'anneau de taille 2k.

Soit $p_1 \ldots p_k$ les processus sur l'anneau de taille k.

Soit $q_1 \ldots q_k\ q'_1 \ldots q'_k$ les processus sur l'anneau de taille 2k.

A la première ronde : On suppose que tous les processus envoient un message.

i = 1 à k -1 q_i envoie un message m à $q_{(i+1)}$ q_k envoie m_k à q_1

i = 1 à k -1 q'_i envoie un message à $q'_{(i+1)}$ q'_k envoie m_k à q_1

Dans une exécution de l'anneau à k :

p_i envoie le même message m à $p_{(i+1) \bmod k}$.

⇒ p_i, q_i et q'_i reçoivent le même message de leur prédécesseur.

A la fin de la première ronde, p_i, q_i, q'_i sont dans le même état.

Soit $I_2 = \{ p_i \mid p_i$ envoie un message dans la 2e ronde $\}$

$X = \{ p_i \mid i \in I_2 \}$

$Y = \{ q_i \mid i \in I_2 \}$

$Z = \{ q'_i \mid i \in I_2 \}$

A la fin de la ronde, p_i, q_i, q'_i, ∀ i sont dans le même état.

L'algorithme A élit un leader sur l'anneau de taille k. ∃ x tel que un processus unique est élu à la fin de la ronde x. Soit p_i ce processus ⇒ q_i et q'_i sont élus.

On ne peut pas non plus calculer la taille.

Théorème : Si la taille est connue, sur un anneau, on peut calculer par un algorithme déterministe où tous les processus terminent une fonction indépendante de l'ordre des paramètres en utilisant un message

Exemple de fonctions utilisables : somme, produit, et,ou,min,max

Théorème : Si la taille n'est pas connue ; il n'existe pas d'algorithmes déterministes où les processus terminent pour calculer une fonction non constante.

Théorème : Si la taille n'est pas connue, il existe un algo déterministe où il n'y a plus de messages qui circulent pour calculer ET , OU

```
var
        result : boolean
        x : boolean entrée

begin :
        result = x
        if (x) then send <yes> to Next
        receive <yes>
        if(result = false) then begin result = true send <yes> to Next
```

f non constante

∃ k tel que ∃ x_1, ..., x_k et ∃ y_1, ..., y_k

$f(x_1, ..., x_k) \neq f(y_1,...,y_k)$

∃ A qui calcule f

Sur un anneau de taille k qui a comme entrée $x_1 \rightarrow x_k$ la chaîne de message avant terminaison est de K_1 (pour $y_1 \rightarrow y_k$ K_2) K = max (K_1, K_2)

Cours n° 6 - 13/02/2017

Algorithme d'élection

Anneau anonyme
Taille anneau connue n
Probabiliste
répéter pour i = 1 à l'infini faire :

- *on tire un nombre au hasard entre [1 ... n] Id[i]*
- *on fait circuler un message avec son identité*
- *quand le message a fait le tour (c'est à dire le compteur est n) alors si :*
  - *v = min des infos de message | s'il n'y a qu'un seul processeur qui a v alors si v = ma val alors élu sinon Perdu.*

// quand n reçoit un message < j, liste, nb > ⟶ envoyer < Id[j], liste + nouvel, nb +1 >
Tous les processus ont la même liste d'identité pour chaque phase (valeur de i)
for i = 1 à ∞ :

- *tirage*
- *vérification*
  si 1 seul ⟶ élu

Propriété : tous les processus s'arrêtent avec la même valeur de phase (i) ou continuent ∞.
Propriété : si les processus s'arrêtent il y a exactement un élu.
Propriété : si dans une phase il y a un processus élu (c'est à dire un seul processus à l'id minimal) alors tous les processus s'arrêtent dans cette phase.

P = la probabilité que si n processus tirent un nombre entre 1 et n au hasard (uniforme) alors il n'y a qu'un processus qui a l'identité minimale.

n^n *tirages* $\frac{1}{n}(n-1)^{n-1}$

$P \geq \frac{1}{n}\frac{(n-1)^{n-1}}{n^n}$ $P > 0$

Quelle est la probabilité que l'algorithme ne soit pas terminé après k phases ? $(1 - P)^k$

Si on termine en k phases combien de messages échangés ? $k \cdot n \times n$

Dans le "bon cas" :

- diminuer le nombre de messages
- diminuer le nombre de phases

Si lors d'une phase un processus n'a pas l'id minimale alors son état = attente et aux phases suivantes il passe le message.

```
var     state   candidat, lost, leader       init candidat
        phase   int                          init 0
        id      int
        stop    boolean              init FALSE
begin
        phase = 1
        id = random ({1 ... n})
        send (< token, phase, id, 1, TRUE >) to Next
        while not stop do :
                receive a message
                message = < token, ph, i, h, b >
                if (h = n) and (state = candidat) then
                        if b then /* élu
                                state = leader
                                send (< ready >) to Next
                                receive (< ready >)
```

```
                        stop = TRUE
                else /* si plusieurs min
                        phase = phase + 1
                        id = random ({1 … n})
                        send (< token, phase, id, 1, TRUE >) to Next
        else /* h != n or state != candidat
                if state != candidat or ph > phase or (ph = phase and i < id) then
                        state = lost
                        send (< token, ph, i, h+1, b >)
                else if (ph = phase) and (i = id) then
                        send (< token, ph, i, h+1, false >)
                else
                        state = lost
                        send (< ready >) to Next
                        stop = TRUE
end
```

Diffusion (Broadcast)

Diffusion fiable (reliable broadcast)

Hypothèse : Une primitive de communication point à point fiable send/receive (pas de corruption, pas de duplication, pas de perte). On suppose que la communication est **asynchrone et qu'on ne connaît pas le temps de diffusion**.
G = (V, E) graphe de communication (fortement) connexe.

[Graphe très moche, qu'on va espérer que Soso va le noter]

Rbroadcast
Rdeliver
Un processus qui Rbroadcast(m) alors tous les processus doivent Rdeliver(m).
Si un processus q Rdeliver(m, p) alors (m, p) a été Rbroadcast par p.
Un processus Rdeliver(m, q) au plus une fois.

[Hors de question que je note ça, Soso !]

```
L = ∅
upon received (m, q) from r
if (m, q) ∉  then
        L = L ∪ {(m, q)}
        Rdeliver (m, q)
        send (m, q) to all except r
```

Diffusion causale (causal broadcast)

G = (V, E) graphe de communication complet.
RBroadcast + Si pour un processus q $Cdeliver_q$ (M) $<_q$ $Cbroadcast_q$ (M') alors ∀ $Cdeliver_p$ (M) $<_p$ $Cdeliver_p$ (M').

R Broadcast(mp,)
        cpt[p]++
        send(m, p, cpt) to all

```
cpt = < o….o > (taille n)
```

```
upon received (m, q, c)
        L = L ∪ {(m, q, c)}
while ∃ (m, q, c') tel que ∃ j c'[j] = cpt[j] + 1 et ∀ k ≠j c'[j] ≤ cpt[j] do
        cpt[q]++
        Rdeliver(m, q)
        L = L - {(m, q)}
```

Diffusion atomique ordonnée (total order broadcast)

RBroadcast + Si pour un processus p Adeliver(M) puis Adéliver(M'), alors ∀ q si q Adeliver(M'), il y a avant Adeliver(M).

Asynchrone : les seuls messages envoyés sont send(M + info) to all lors du ABroadcast.
⇒ Impossible

[On va espérer que Soso suit toujours...]

Cours n°7 : 20/02/2017

Le but est de montrer qu'il n'y a pas de pertes de messages, ou de montrer que les pertes de messages sont limitées.
L'histoire c'est ***l'attaque coordonnée***. L'idée c'est qu'on a une ville, que l'on veut attaquer, et on a des généraux qui ont un avis sur l'attaque (1 j'attaque, 0 j'attaque pas) et ils doivent se mettre d'accord sur si on attaque ou non. Les généraux vont communiquer entre eux pour se mettre d'accord sur l'attaque via des messagers. La ville se défend en envoyant des soldats qui tuent les messagers ⇒ les messages peuvent se perdre.

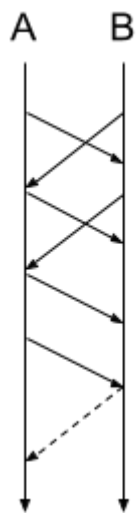
Entrée 0 ou 1
Sortie 0 ou 1
// On montre d'abord pour 2 processus avant de généraliser
On doit vérifier les trois propriétés suivantes :

Accord: Si 2 processus décident, alors ils décident la même chose - soit ils attaquent, soit ils n'attaquent pas.

Validité : Si tous les processus commencent avec 0, la seule décision possible est 0. Si tous les processus commencent avec 1 et que **tous les messages sont reçus** alors la seule décision possible est 1.

Terminaison : Tous les processus décident.

Hypothèse : perte de messages.



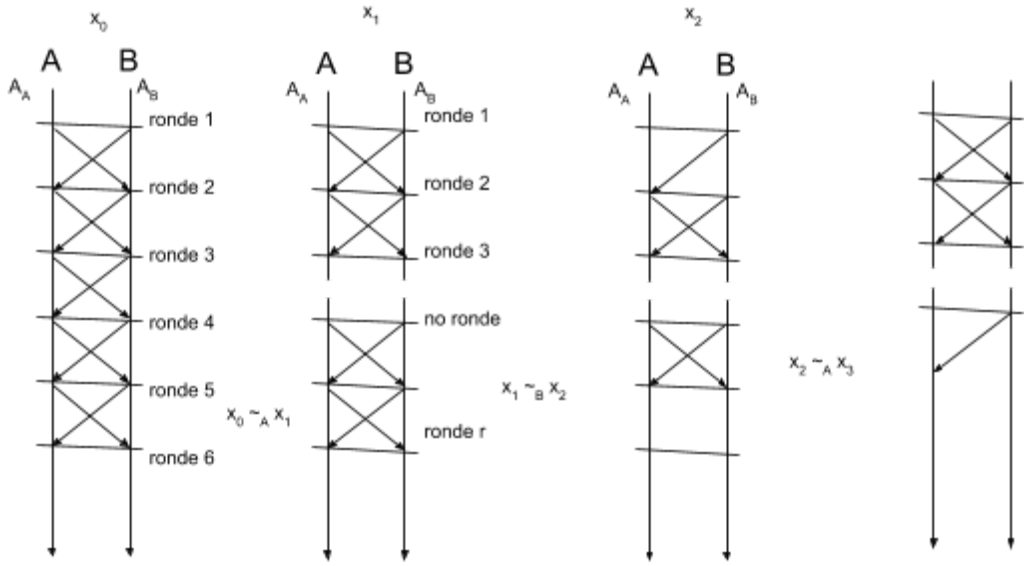
Système synchrone.
Une exécution x
y l'exécution x

- le dernier message à la ronde r de B vers A,
- aux rondes suivantes tous les messages se perdent

On se place du point de vue de B, il a exécuté son algo, il a envoyé et reçu des messages. Du point de vue de B, x et y sont indistinguables. Dans cette exécution on enlève le dernier message envoyé de B vers A et après, tous les messages envoyés

sont perdus. A peut faire d'autre chose, mais il ne recevra plus rien de B et tous ses messages seront perdus. Du point de vue de B, ça échoue.

On suppose qu'il existe un algorithme $A_A A_B$ qui résout le problème. On considère l'exécution x_0 où A et B ont pour valeur initiale 1 et où il n'y a pas de perte de messages.



Il existe une ronde telle que A et B ont décidé. On suppose l'exécution $x_1 = x_0$ - le message de A vers B à la ronde r-1 à toutes les rondes >= r, tous les messages se perdent.

- On regarde l'algo du côté de A, il a reçu la même valeur. Du point de vue de A $x_0 = x_1$.
- Du côté de B, ce sera pas la même chose, vu que les messages sont perdus, on sait pas s'il envoie d'autres choses. B doit décider, et doit décider la même chose que A pour terminer (selon les propriétés du dessus).

On considère $x_2 = x_1$ - le message de B vers A à la ronde r-1 et a toutes les rondes >= r ; tous les messages se perdent.

- Du point de vue de B, l'exécution x_1 et x_2 sont équivalentes.
- Cependant, du point de vue de A, on se retrouve dans le même cas que B avec x_1, il faut choisir pour terminer, et il faut choisir 1.

Conclusion de tout ce bordel : Un tel algorithme n'existe pas.
Problème : Attaque coordonnée
Hypothèse : pertes de messages + synchrone.
On va essayer d'enlever des hypothèses:
Peut on le faire si c'est asynchrone?

Nope même en asynchrone on ne peut pas le faire.

L'attaque coordonnée c'est A et B sont d'accord sur la réception des messages.
//Insert dessin protocoles réseaux here

Il faudrait utiliser des accusés de réceptions.

Algorithme Probabiliste

On fixe r et on veut qu'au bout de r rondes les processus décident.
[j'ai la flemme de refaire des flèches]

Même valeur à chaque k $\in$ {1, ..., r} ronde.
A la ronde x > k, k=1 pour A ?? 1 pour B alors D1 sinon D0

init niveau = 0 $v_a = \perp, v_b = init$ pour B
$v_b = \perp, v_a = init$ pour A

Code de B

Un processus envoie à chaque ronde
- sa valeur init
- son niveau

quand il reçoit un message < w, l > si $v_a = \perp$, alors v_a = w, niveau = l + 1

Code de A

Le même que pour B, mais on remplace v_a par v_b.

Propriété : $\forall$ r, $niveau^r_A = niveau^r_B + \varepsilon$ avec $\varepsilon = 0,1,-1$

Par récurrence :
H(r) = $\forall$ r' <= r $niveau^{r'}_A = niveau^{r'}_B + \varepsilon$ avec $\varepsilon = 0,1,-1$

H(1) $niveau^1_A$ (niveau de A à la ronde) 1 = $niveau^1_B$ avec $\varepsilon = 0,1,-1$
0 message ou chaque processus a reçu 1 message $niveau^1_A = niveau^1_B$
Si A a reçu un message et pas B, $niveau^1_A = niveau^1_B + 1$

Un schéma de communication S décrit pour chaque ronde de 1 à r les messages reçus.
Un adversaire = un schéma de communication + les valeurs initiales des processus.

Algorithme $\rightarrow$ 1 tirage aléoatoire
la valeur du tirage = adversaire a chelou $\rightarrow$ Exécution

Pour [A chelou] donné, l'ensemble des tirages me donnent un ensemble d'exécution de l'algorithme, i.e. une distribution probabiliste sur l'ensemble des exécutions.
$Pr^{A\ chelou}$ si k val différente du tirage aléatoire :

$Pr^{A\ chelou}(X)$ = 1 / k (card { x | l'exécution avec le tirage x qui vérifie X }

On modifie les propriétés d'accord. A adversaire [A chelou]
$Pr^{A\ chelou}$ (1 processus décide 0 & 1 processus décide 1) <= ε
Validité + Terminaison

Code de l'algo :
n = 1 / ε
init niveau = 0
v_A = valeur init
$v_B = \perp$
A tire un nombre aléatoire entre 1 et r (pas B)
for ronde = 1 à r

send (v_A, niveau) to B
upon receive (v, n) fro, m B
if $v_B = \perp$ then v_B = v
niveau = n + 1

Pour B :
for ronde = 1 à r
send (v_B, niveau, K) to A
upon receive (v, n, α) from A
if $v_A = \perp$ then v_A = v, K = α
niveau = n + 1

if niveau >= K and $v_A = v_B = 1$ then output = 1 else output = 0

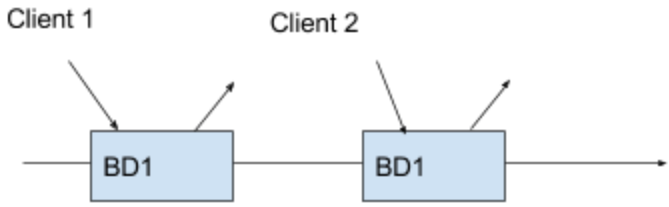
Terminaison
Validité
Si tous les messages reçus $niveau^r_A = niveau^r_B = r$
Accord
$niveau^r_A >= K + 1$
⇒ $\underline{niveau^r_B >= K}$
$niveau^r_A <= K - 1$
⇒ $niveau^r_B <= K$
Seuls cas de désaccord :
$niveau^1_A = K$ && $niveau^1_B = K - 1$
et $niveau^1_A = K - 1$ && $niveau^1_B = K$

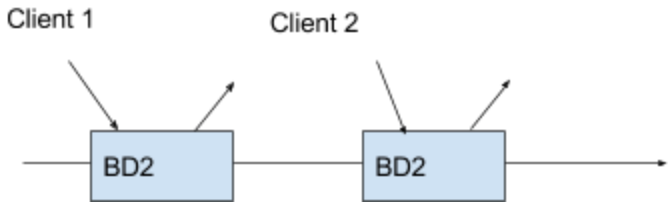
Cours n°8 - 27/02/2017

Tolérance aux pannes
Pannes de processus

- **pannes par arrêt** : il ne fait plus rien, plus d'envoi de messages, panne relativement bonne, ça ne pose pas plus de problèmes que ça. Parmi ses pannes, on a différents groupes :
  - pannes par omission de réception : on "oublie" de recevoir les messages
  - pannes par omission d'envoi : on "oublie" d'envoyer les messages
  - les 2
- **pannes byzantines** (les plus compliquées à traiter):
  - pannes intentionnelles
  - pannes non intentionnelles

Quand on a une application distribuée, on essaye de maintenir un état.



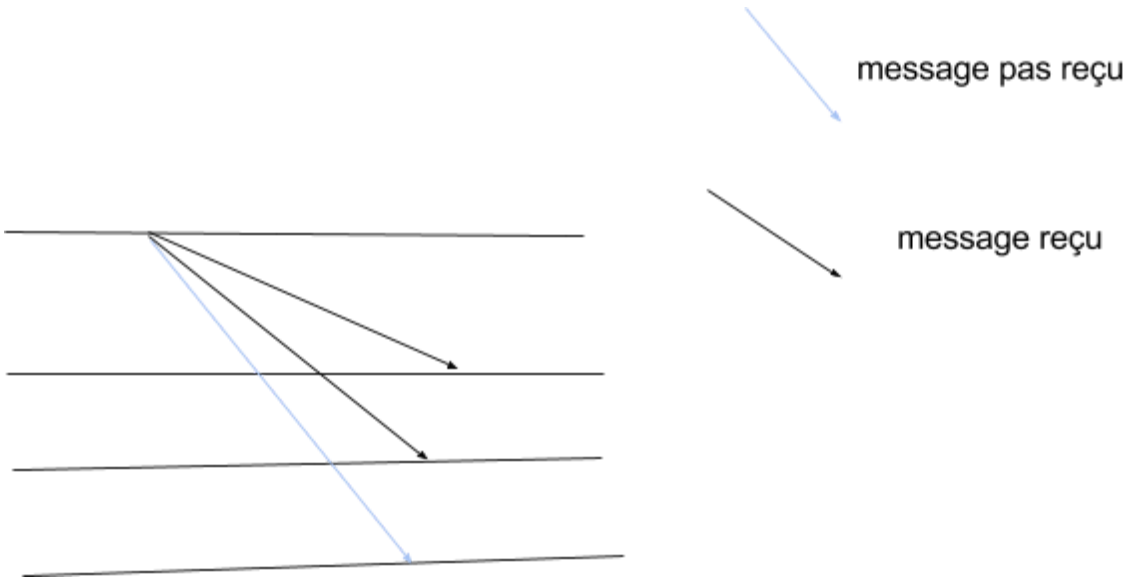


Pour tolérer les pannes, on va faire de la réplication, si on a un des 2 qui crash, on a le deuxième qui survit le temps que l'on redémarre le processus, ainsi on a toujours accès à la BDD.
Il existe 2 types de réplications : Active, ou Passive

- **Passive** : Maître/esclave, si le maître crash, l'esclave sert de backup et devient le maître. Après son redémarrage, l'ancien maître deviendra esclave.
- **Active** : Les processus font les mêmes opérations .

Quand on fait de la réplication, on veut que les BDD soient cohérentes entre elles. Dans le cas d'une vente de billets, on ne veut pas vendre 2 fois la même place, selon l'ordre de traitement par les processus. On instaure donc que les demandes des clients sont traitées dans le même ordre. Le client va prendre n'importe quelle réponse qui lui arrive. Ainsi dans le cas de la réplication, il faut qu'on ait les mêmes réponses en sorties.

En utilisant l'Atomic Broadcast (vu précédemment), on assure que les requêtes clients arrivent dans le même ordre sur tous les processus. En considérant une panne par arrêt, supposons qu'on a un processus qui envoie un message m à tous les autres processus. S'il s'arrête, tous les processus n'auront pas les mêmes informations (voir schéma ci dessous).



Consensus : chaque processus propose une valeur et décide d'une valeur (décision irrévocable). Il est défini par 3 propriétés :

- **Accord** : Si 2 processus décident une valeur, c'est la même
- **Intégrité** : Si un processus décide d'une valeur, cette valeur a été proposée.
- **Terminaison** : Tous les processus corrects décident

Atomic BroadCast ⇔ Consensus (avec du Reliable BroadCast, Diffusion fiable)

Les algo de consensus, permettent de faire de la réplication et que les processus s'entendent sur une valeur.

Système synchrone, avec une notion de ronde :

- Les processus émettent au début de la ronde
- A la fin de la ronde, ils ont reçu tous les messages
- Ils calculent leurs nouveaux états

Sans panne, tous les processus reçoivent le même nombre de messages : $M_p^r = M_q^r$
Avec panne, il va être possible que le nombre de message reçu soit différents : $M_p^r \neq M_q^r \Rightarrow m \in M_p^r$ et $m \notin M_q^r$

Pannes par arrêt : x est en panne ⇒ x n'émettra plus aucun message après la ronde r.

Un processus est correct s'il émet un message à toutes les rondes (incorrect sinon). SI un processus est incorrect : soit r la première ronde dans laquelle un de ses messages n'a pas été reçu ⇒ aucun message ne sera reçu aux rondes suivantes.

Un processus incorrect tombe en panne à une ronde :

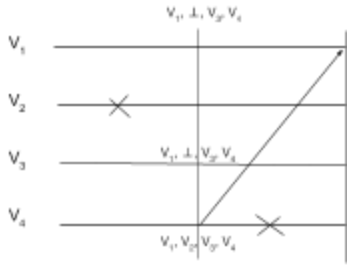
- n processus
- t pannes possibles /tolérées

⇒ Ce processus ne fonctionnera que si le nombre de pannes est ≤ t

Un protocole résout un problème P en tolérant t pannes par arrêts ⇔ dans toutes exécutions où il y a moins de t panne (pannes effectives ≤ t). L'exécution satisfait la spécification de P.

```
Code de pi = V[i] = proposition
i ≠ j v[j] = ⊥
for x = 1 to t+1 do :
        /*r : ronde */
        send(V) to all
        w={(w)/w est reçu pendant la ronde }
        If ∃k st V[k] = ⊥ and ∃ w ∈ W st w[k] = ⊥ then
                V[k] = w[k]
        d=maxV[j] /*j∈{1,...,n}
        decider d
```

L'algorithme résout le consensus en tolérant t pannes (t < n)
t = 1



***Preuve** :*

- ***Terminaison*** : oui
- ***Intégrité*** : pour pi V[i] ‡ ⊥

∀ k si V[k] ‡ ⊥ alors V[k] est proposé pour pk

- ***Accord*** (t pannes au plus)

Dans t+1 rondes, il y a une ronde sans pannes. Soit x celle ci,

Lemme : ∀ p,q vivants à la fin de la ronde x, alors (Vp)^x = (Vq)^x

Démonstration du lemme :

Par contradiction, ∃ y tel que (Vp)^x[y] = ⊥

(Vq)^x[y] = ⊥

(1) si (Vq)^x-1[y] != ⊥
Alors à la ronde x q → p donc p a reçu ce message et (Vq)^x[y] != ⊥ =====> Contradiction

(2) si (Vq)^x-1[y] != ⊥, q a appris Vq[y] lors de la ronde x pour un message de z, comme il n'y a pas de pannes à la ronde x, et a aussi reçu le message de z. =====> Contradiction

Lemme 2 : ∀ r ≥ x ∀ p,q

(Vp)^x = (Vp)^r = (Vq)^r

A la ronde t+1, tous les processus vivants ont le même vecteur.

n processus

t pannes tolérées

f pannes effectives

min (f+2, t+1,n-1)

Théorème : Tout algorithme qui résout le consensus en tolérant t pannes a au moins une exécution en t+1 rondes.

Preuve : On considère un système synchrone S où il y a au plus 1 panne par ronde. Consensus binaire 0/1.

Par contradiction : Je suppose qu'il existe un algo A qui résout le consensus en tolérant t pannes aux t rondes.

Configuration à la fin de chaque ronde :

- état du processus vivant
- crash pour les autres

Une exécution partielle rk est une exécution de l'algo A dans S jusqu'à la fin de la ronde k.

Ck la configuration à la fin de la ronde k.

dessin chelou

Si à partir de Ck en exécutant A dans S un processus décide 0, Ck est 0-valante ou décide 1, Ck est 1-valante
Ck est bi-valante si Ck est 0-valante et 1-valante, mono-valante sinon.

dessin chelou presque pareil que le précédent

Lemme 1 : Toute exécution partielle de (t-1) ? r_{t-1} est monovalente
Preuve : Par contradiction

exactement le même dessin

On est dans S. Il peut encore y avoir 1. À partir de r_{t-1} on fait une ronde sans panne r^0
Dans cette ronde r^0 les processus décident 0

Il y a une exécution seule de r_{t-1} où les processus décident 1 : r^1

dessin avec des étoiles sataniques

Soit p le processus en panne à la ronde t (voir dessin pas fait)

Lemme 2 : Il existe une configuration initiale bivalente

1	1	1	...	1
0	1	1	...	1
0	0	1	...	1
0	0	0	...	1
0	0	0	...	0

Lemme 3 : Il y a une exécution partielle de (t-1) rondes bivalentes
Preuve par induction HR(k)
$0 < k < t-1$, il y a une exécution partielle bivalente de k rondes
$HR(k) \Rightarrow HR(k+1)$

retour du dessin satanique

$r_t \rightarrow r_t$
On ajoute à r le message que p n'a pas envoyé à un processus

petit dessin sympathique

Accord : Tous les processus qui délivrent un message délivrent le même message
Terminaison :Tous les processus corrects délivrent un message
Validité : Si s est correct et diffuse m alors tous les processus corrects délivrent m
Intégrité : Un processus délivre un message au plus une fois. Si un processus délivre m alors, soir m = SF ou m a été diffusé par l'émetteur.

TRB ⇔ consensus
On a du consensus, p est l'émetteur pour TRB pour diffuser m à la ronde p envoie m à tous.
1ère ronde :
q !=p
q a reçu(m) de p ou dans consensus vq = m, q n'a rien reçu de p. dans ce cas vq = SF exécute le consensus avec vq comme val proposée.
La valeur décidée est la valeur délivrée
consensus ⟶ TRB

TRB ⟶ consensus

Cours n°9 du 06/03/2017

Pannes byzantines

Une panne byzantine est une panne où le processus envoie des trucs qu'il ne devrait pas envoyer. On s'intéresse aux problèmes de consensus pour faire de la réplication active. Du coup, avec la panne byzantine il peut recevoir n'importe quoi. Si on a une panne byzantine, il faut attendre la réponse de minimum 3 serveurs (si on attend une ou deux réponses, on ne pourra pas distinguer une bonne réponse d'une mauvaise réponse).

Accord : Si deux processus corrects décident de la même valeur.
Terminaison : Tous les processus corrects décident.
Validité : Si tous les processus corrects proposent la même valeur v alors aucun processus correct ne décide une valeur différente de v.

⇒ Si on a t pannes, il faudra 3tprocessus pour pouvoir faire la réplication active.
$Specif \Rightarrow n > 2t$
$Si\ n = 2t\ (\leq)$

$|P| = |Q| = n$ (*exécution avec P et Q corrects*)

P est byzantin ⇒ La valeur décidée aurait dû être 1.

TRB

broadcast(m) / deliver(m)
diffuse(m) / delivre(m)
Un processus particulier s

On reprend le même problème
Validité : Si s est correct, et diffuse m, alors s délivre m.
Accord : Si un processus correct délivre m, alors tous les processus corrects délivrent m.
Terminaison : Tous les processus corrects délivreront un message.
Intégrité : Un processus correct délivre au plus un message et ils délivrent m != SF (Sender Faulty). Mais peut-être qu'il a envoyé des choses, mais qu'il n'a pas diffusé un message - donc la seconde partie de la condition (en orange ici) n'est pas prise en compte. Il n'y aura pas de message spécial Sender Fault.

Equivalence entre consensus et TRB

⇒ Si on nous donne un problème, alors on est capable de résoudre l'autre.

Est-on capable de faire du TRB dans le cas byzantin en partant du consensus ?
On a un processus particulier nommé s.
1ère ronde

- Pour diffuser(m), s envoie m à tous les processus.
- Sur réception d'un message par p , prop = m sinon prop = 0.

Autre rondes

- Algorithme de consensus avec la prop
- A la fin de l'algorithme de consensus, le processus décide x.

⇒ La valeur délivrée est x.

Je veux garantir la **validité** ⇒ si l'émetteur est correct, tous les processus corrects vont recevoir son premier message et proposer ce message là au consensus. La **validité** nous dit que si tous les processus corrects décident une valeur v, alors cette valeur est diffusée. Si s est correct et diffuse m, alors m sera délivré. Si un processus correct délivre m, alors tous les processus corrects délivrent m : si un processus correct délivre un message alors il a décidé ce message, le consensus assure que si deux processus corrects décident, alors ils décident la même valeur.

Si j'ai du consensus, je peux faire du TRB (*si le consensus est en R ronde, alors le TRB sera en R+1 rondes*).

Peut-on faire du consensus à partir du TRB ?
Il faut que n > 2t.
Dans le consensus, chaque processus a une proposition (chaque processus va être émetteur et va diffuser sa proposition).

Algorithme : un processus diffuse (prop) et attend d'avoir délivré un message pour chaque processus. Le TRB assure que si l'émetteur est correct, on délivrera quelque chose un jour. On décide la valeur majoritaire.

Vérification des propriétés :

- **Accord** : chaque processus se retrouve avec le même ensemble de message, donc ils auront la même valeur décidée.
- **Terminaison** : L'attente ne peut pas être infinie, chaque émetteur délivrera le message. On a bien une terminaison.
- **Validité** : tous les processus corrects auront la même valeur de décidée, on aura bien une valeur majoritaire qui sera décidée.

Si on a R rondes en TRB, on aura R rondes en consensus.

Les deux problèmes sont donc équivalents.

p → q
p envoie un message m à q, q sait que p lui a envoyé un message (on ne ment pas sur son identité).
Par contre, p peut envoyer un message a à q et un message b à r.
On peut rajouter de l'**authentification**, c'est à dire signer ses messages : p envoie un message a à q et b à r (c'est toujours possible), mais si p dit à q que x lui a envoyé un message b, il ne pourra pas dire que x lui a envoyé un message a à r : on ne peut pas mentir sur le message qui a été envoyé. Cela permet de distinguer lequel est byzantin sur les deux. Si on dit que x nous a envoyé un message, on envoie un message **signé**. La signature va permettre que quand un processus répète quelque chose, il ne répète pas n'importe quoi. Le processus byzantin va pouvoir ne pas envoyer de message, et répéter quelque chose de faux qui donnera la possibilité de le distinguer.
Ce processus d'**authentification** peut être fait grâce aux clés privées / clés publiques. On va donc pouvoir diminuer le pouvoir de nuisance du processus byzantin (sans parvenir à l'éliminer totalement).

On va montrer qu'on peut faire du **TRB avec de l'authentification** (*Algorithme de consensus avec authentification si n > 2t*) ⇒ consensus avec authentification possible si et seulement si n > 2t.
On fera ensuite un algorithme de **TRB sans authentification (avec n > 3t)** (*Algorithme de consensus si n > 3t*). On montrera ensuite que si n ≤ 3t il est **impossible d'avoir un algorithme de consensus sans authentification** ⇒ Algorithme de consensus sans authentification possible si et seulement si n > 3t.
Authentification : mécanisme de signature.

m	message
p	processus
m:p	message m signé par p

(a) si un processus q reçoit de p un message m:p, alors q peut extraire m et p.
(b) si p est correct, alors aucun processus ne peut envoyer un message contenant x:p (sauf si p a envoyé x:p dans le passé) ⇒ La signature est **infalsifiable**

a) ⇒ m:p1:p2:p3...pk
→ Un processus peut extraire m et p1,...,pk ⇒ n peut extraire chacun des éléments de la chaîne.
b) ⇒ un processus ne peut pas signer un message à la place d'un processus correct. Par contre, les byzantins peuvent s'entendre entre eux et signer les messages des autres byzantins.

à la t+1 ronde
m:s:p1:p2:p3…:pr

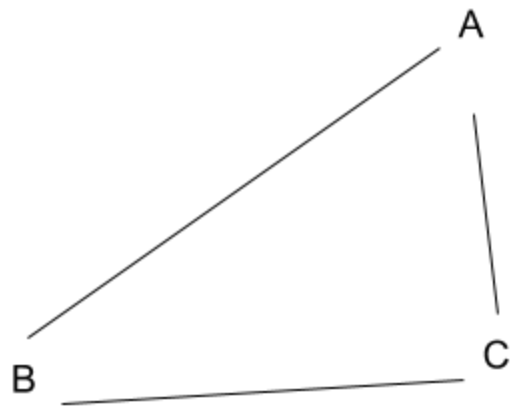
Code de p :
Initialement : si p = s et veut diffuser m
extracted = relay = { m }
sinon extracted = relay = ∅
pour a == 1 à t+1 faire
envoyer (relay:p) si relay != ∅ { x:p | x ∈ relay }
recevoir les messages de la ronde (V cet ensemble de messages)

```
relay = ∅
pour chaque m:p1:p2:...:pr ∈ V faire
        si m ∉ extracted alors
                extracted = extracted U { m }
                relay = relay U { m }
si n = t+1 alors ∃ m extracted= { m }
        alors deliver(m)
sinon deliver(SF)
```

Définition : un processus extrait m à la ronde i si p insère m dans extracted à la ronde i. Par convention, s extrait m à la ronde 0.

Lemme : Si un processus correct extrait m, alors tous les processus corrects extraient m.
Soit i la 1ère ronde à laquelle un processus correct extrait m.
si i = 0 ⇒ tous les processus extraient m à la ronde i (il y a au moins 1 ronde car t+1>= 1)
Si i!=0 ⇒ p extrait m à la ronde i
p a reçu m:p1:p2:....:pi ⇒ p...pi sont byzantins (car on peut extraire qu'à partir de i).
S'ils sont fautifs, j'en ai au plus t, alors i <= t.
L'algorithme fait t+1 rondes ⇒ p enverra à tous les processus m:p1....pi:p à la ronde i+1 (<=t+1)
⇒ tous les corrects extraient m.

(Sans authentification) Il est impossible de faire du consensus si n<=3t.
t = 1
n = 3
Il existe un algorithme qui fait du consensus.



Cas général :
On suppose qu'il existe un algorithme $\mathscr{A}$. Soit A, B, C trois ensemble de processus de taille t.
p simule les processus de A en donnant comme proposition sa propre proposition.
q simule les processus de B
r simule les processus de C
$\mathscr{B}$ est un algorithme de consensus pour 3 processus qui tolère 1 panne byzantine.

On a montré qu'un tel algorithme n'existait pas.

Broadcast authentifié : un processus p qui va diffuser un message m à la ronde k. Un processus accepte le message (p, m, k).

Correction : si un processus correct diffuse (p, m, k) dans la ronde k, tous les processus corrects acceptent (p, m, k) dans la ronde k.

Non forgeabilité : Si un processus p est correct et ne diffuse pas (p, m, k), alors aucun processus correct n'acceptera (p, m, k) à aucun ronde.

Relai : Si un processus correct accepte (p, m, k) dans une ronde n ≥ k, alors tous les processus corrects accepteront (p, m, k) au plus tard à la ronde n+1

<u>Cours du 13/03/2017 : THE LAST ONE</u>

- Byzantine : 3t < n
- Synchrone
- Pas d'authentification
- TRB 0 ou 1

Broadcast authentifié : Un processus diffuse un message m dans la ronde k : broadcast(p, m, k). Un processus q accepte le message (p, m, k) avec la spécification suivante.

Correction : Si un processus correct diffuse (p, m, k) alors tous les processus corrects acceptent (p, m, k) à la même ronde.

Relai : Si un processus correct accepte (p, m, k) alors tous les processus corrects acceptent (p, m, k) au plus tard à la ronde r+1.

Non forgeabilité : Si un processus correct ne diffuse pas (p, m, k) alors aucun processus correct ne l'acceptera à aucune ronde.

1 ronde (k) en 2 phases (2k-1 et 2k)

Code de p

→ phase 2k-1 : Pour broadcast (p, m, k)

```
send (init, p, m, k) to all
```

→ phase 2k :

```
if received (init, q, m, k) from q in phase 2k-1 then
        send (echo, q, m, k)
// si pas déjà fait, alors :
 if received (echo, q, m, k) from at least (n-t) distinct processus in phase
2k - 1 then
        accept (q, m, k)
```

Ronde r > k

Phase 2r-1, 2r

if at least n-2t message (echo, q, m, k) received and not yet sent (echo, q, m, k) then

send (echo, q, m, k)

if at least (n-t) ... et if not yet accepted

[Dessin moche]

Correction :

Lemme : Si un processus correct envoie (echo, p, m, k) alors p doit avoir envoyé (init, p, m, k) à au moins un processus x correct en phase 2k - 1.

⇒ **Démonstration :**

l : la 1ère phase pendant laquelle un processus correct envoie (echo, p, m, k)

Soit q un de ces processus.

- *Soit l > 2k :* q a reçu n-2t echo;

  $n - 2t \geq t + 1 \Rightarrow$ reçu d'un correct

  ⇒ Contredit la définition de l, donc l = 2k ⟶ q a reçu à la ronde 2k-1 init de p.

Non forgeabilité :

Si p est correct et ne fait pas de diffusion de (p, m, k) :

⇒ il n'envoie pas (init, p, m, k) à la ronde k.

Si un processus correct accepte (p, m, k) :

⇒ a reçu n-t messages echo

⇒ a reçu n-2t au moins qui proviennent de processus corrects

⇒ un processus correct au moins à envoyé echo

⇒ p doit avoir envoyé (init, p, m, k) par le lemme.

Relais :

Si un processus q correct accepte (p, m, k) durant la phase i, avec i = 2r-1 ou i = 2r (à la ronde r), alors q doit avoir reçu n-t message echo (p, m, k) à la phase i

⇒ n-2t de ces messages viennent de processus corrects

n-2t processus corrects ont envoyé (echo, p, m, k) à la phase i ou avant.

⇒ tous les processus corrects ont reçu n-2t echo à la phase i ou avant

⇒ envoyer (echo, p, m, k) à la phase i+1 ou avant

⇒ tous les processus corrects ont reçu au moins n-t messages echo à la phase i+1 ⇒ ronde r ou r+1

Code de p TRB du message m ({0, 1}) par l'émetteur

si p est l'émetteur alors val = m, sinon val = 0

pour r = à à t+1 faire :

si val = 1 et p n'a pas diffusé de message avant, alors

broadcast authentifié ⇒ broadcast(p, 1, r)

si dans les rondes r' ≤ r p a accepté $(p_k, 1, r_k)$ de r processus distincts incluant l'émetteur alors

val = 1

deliver (val) (pour le TRB)

Preuve TRB :

Terminaison : Tout processus correct délivre un message

Intégrité : On délivre un message au plus une fois

Validité : Si l'émetteur est correct et broadcast(m) alors deliver(m)

Accord :

Par contradiction, supposons qu'il n'y a pas accord.

Soit r la 1ère ronde où val = 1, pour un processus correct p.

Cas où r < t+1 : à la ronde r+1, p broadcast(p, 1, r+1) et tous les corrects acceptent (p, 1, r+1)

Tous les corrects acceptent r messages dont (s, 1, o) au plus tard à la ronde r+1.

r = t+1

p a eu t+1 accept

⇒ 1 des accept vient d'un correct
1 correct à broadcast (p,1,x) a une ronde x <= t+1
⇒ sa valeur de val a été modifiée à x-1 à une ronde avant r
⇒ Contradiction avec la définition de r.

Asynchrone

Pas de perte de message, communication fiable.

- Diffusion fiable
- Diffusion totalement ordonnée
  (Atomic Broadcast)
  (Diffusion fiable)
  consensus

Par contradiction : A qui permet de faire du consensus en tolérant t pannes par arrêt.
n processus est décomposé en 2 ensemble A et B
| A | ≤ t
| B | ≤ t
A ∪ B est l'ensemble des processus

1ère exécution : tous les processus de A proposent 0
tous les processus de B morts depuis le début (ils n'envoient aucun message)
∃ α ou tous les processus de A ont décidé (**Terminaison**)
(**Accord**) même valeur
(**Intégrité**) 0

2e exécution : tous les processus de A initialement morts
tous les processus de B sont corrects et proposent 1

Tous les processus sont corrects

- processus de A proposent 0
- processus de B proposent 1
- tous les messages A → B et B → A mettent max(α , β)

Pour A, 1re et 3e exécution sont indistinguables
Pour B, 2e et 3e exécution sont indistinguables
→ Les processus de A décident 0
→ Les processus de B décident 1
contredit l'accord du consensus

Théorème : Il n'y a pas d'algo déterministe de consensus tolérant 1 panne en asynchrone (FLP 85)

Du coup soit on fait un algo probabiliste soit un algo partiellement synchrone.
Pourquoi on fait de l'asynchrone?
Le synchrone c'est très dur à assurer sur un système, il faut qu'il y ait des contraintes fortes sur ce système.
Le but du partiellement synchrone, on va profiter des moments ou ça marche bien pour le consensus, et essayer de pas foutre la merde quand ça marche plus.

En faisant un consensus probabiliste ou partiellement synchrone, on veut garantir l'accord, l'intégrité. Du coup, ici la terminaison sera probabiliste ou partiellement synchrone.

Dans ce cas la, on a l'algo Paxos (déterministe), on a un chef, et c'est lui qui va lancer les consensus à la suite. Si le chef est byzantin c'est la merde.

Algorithme probabiliste pour du consensus binaire (BenOr)

- fonctionne seulement si n > 2t
- tirage uniforme 0/1

Les rondes s'arrêtent en fonction du nombre de messages reçus.

```
r = 1
pour toujours
        envoyer (prop, P, r) à tous
        attendre de recevoir n-t messages (0/1,P ,r)
        si tous les messages sont (0,p,r) (respectivement (1,p,r)) alors w = 0 (resp w=1)
        sinon w = ?

        //2eme ronde
        envoyer(w,D,r) à tous
        attendre de recevoir n-t messages ( 0/1/?, D,r)
        Si tous les messages sont (0,D,r) ⟶  décide 0
        Sinon   si tous les messages sont (1, D, r) ⟶ décide 1
        Sinon   si ∃ (v, d, r) avec v !=  ? alors prop = v
                sinon prop = tirage aléatoire
```