Au sujet des paramètres dans les systèmes répartis

Acadie

29 janvier 2017

plan

1. UN paramètre numérique
   - Élection
   - Exclusion mutuelle
   - Détection de la terminaison
2. DEUX paramètres numériques
   - Le problème du consensus
   - Système synchrone
   - Système asynchrone
3. TROIS paramètres numériques
4. Paramètres non numériques
   - Modèles
   - Détecteur de fautes

Plan

1. UN paramètre numérique
   - Élection
   - Exclusion mutuelle
   - Détection de la terminaison
2. DEUX paramètres numériques
   - Le problème du consensus
   - Système synchrone
   - Système asynchrone
3. TROIS paramètres numériques
4. Paramètres non numériques
   - Modèles
   - Détecteur de fautes

Nombre de processus

1 paramètre = nombre de processus

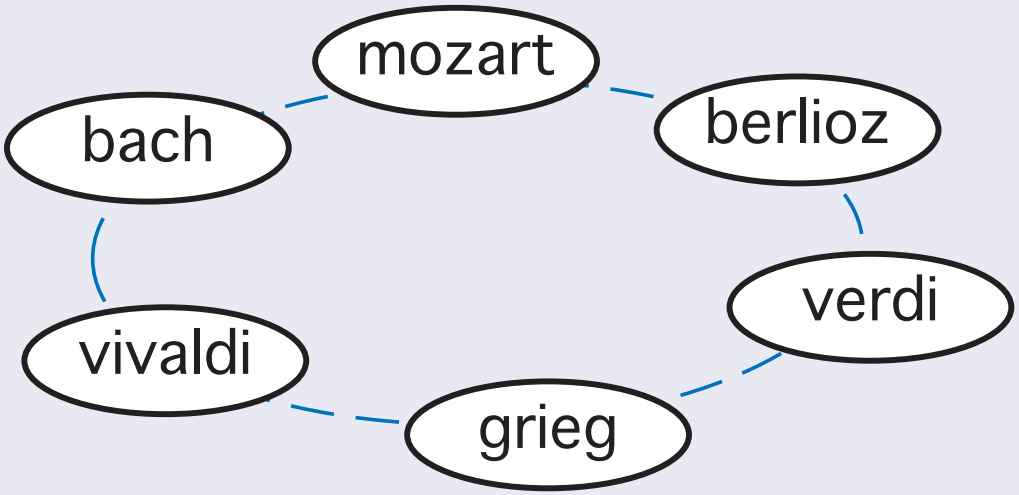
Opérations utilisées sur le processus i :

- $pick\ j \neq i$
- $i+1$, $i-1$, $i = N$, $i \neq 0$
- $(i+1)\%N$, $(i+N-1)\%N$ (anneau logique)
- $i\%2 = 0$
- $SUBSET\ 1..N$

Exemple : l'élection

Le problème de l'élection

Objectif : Élire un seul processus



- Un processus a une identité unique qu'il connaît
- Un processus ne connaît pas le nombre global de processus
- Un processus ne connaît pas l'identité des autres
- Communication sur un anneau logique

Solution qui conduit à l'élection du plus petit

```
Process P(id : 0..N-1) {
  type Etat = {candidat,élu,perdant};
  Etat étatCourant ← candidat;
  on start :
      send Candidat(id) to P[id⊕1]; // chacun candidate
  on reception Candidat(proc) from P[id⊖1]:
      if (proc < id) send Candidat(proc) to P[id⊕1];
      else if (proc = id) étatCourant ← élu;
      else nop;  // ignorer le message
  on (étatCourant = élu) :
     send Elu(id) to P[id⊕1];
  on reception Elu(proc) from P[id⊖1]:
     if (proc ≠ id) then
          étatCourant ← perdant;
          send Elu(proc) to P[id⊕1];
     endif
     terminate
```

Algorithme d'exclusion mutuelle à base de permission

Un processus candidat doit demander aux autres processus la permission d'entrer en exclusion mutuelle

Hypothèses

- Chaque processus connaît l'identité des N processus
- Réseau de communication fiable et maillé (non FIFO)
- Pas de défaillance de processus

Solution

- Permissions individuelles en demandant à tous l'autorisation
- Ordonnancement des requêtes par datation

Messages

- Message de requête : le site i demande l'autorisation à j
- Message d'autorisation : le site j donne son autorisation à i
- Pas de message de refus : le refus est temporaire

Algorithme de Ricart et Agrawala

Principes

- Requêtes d'entrée totalement ordonnées :
  $\Rightarrow$ Utilisation d'horloges de Lamport
- Chaque processus P_i candidat ou en exclusion connaît la date de sa requête courante $Date(R_i)$
- Un candidat entre en exclusion s'il a obtenu les permissions de tous les autres
  $\Rightarrow$ il possède alors la requête la plus ancienne
- Revient à vérifier que la requête R_i d'un processus P_i est la plus vieille requête des processus candidats ou en exclusion :
  $\forall k : P_k.Etat \neq hors \Rightarrow Date(R_i) \leq Date(R_k)$

Algorithme de Ricart-Agrawala

```
Process P(i : 0..N-1) {
 type Etat = {hors,candidat,exclusion}; Etat EC ← hors;
 Date hloc ← new Date(i,0); // horloge locale
 Date dr; // date de la requête de ce site
 Set<0..N-1> Att ← ∅;// sites lui ayant demandé l'autorisation
 Set<0..N-1> D;        // sites dont i attend l'autorisation
 on (EC = hors) : // hors → candidat
       EC ← candidat;
       D ← 0..N-1 \ {i};
       dr ← hloc.Top();
       for k ∈ D : send Request(i,dr) to P_k;
 on reception Request(p, d) :
       hloc.Recaler(d);
       if (EC ≠ hors ∧ dr < d) then Att ← Att ∪ {p};
       else send Perm(i) to P_p;
 on (EC = candidat) ∧ reception Perm(p) : // candidat → excl
       D ← D \ {p};
       if (D = ∅) EC ← exclusion;
 on (EC = exclusion) : // exclusion → hors
       for k ∈ Att : send Perm(i) to P_k;
       Att ← ∅; EC ← hors;
```

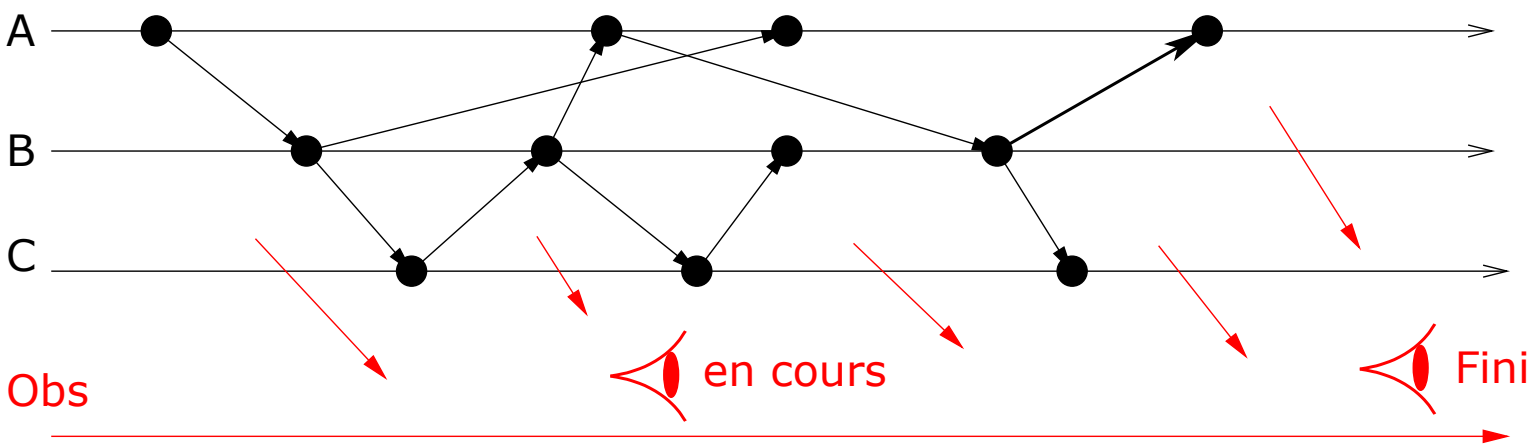
Détection d'une propriété stable

Spécification

- Propriété stable à détecter :

Tous les processus sont passifs ET pas de message en transit.

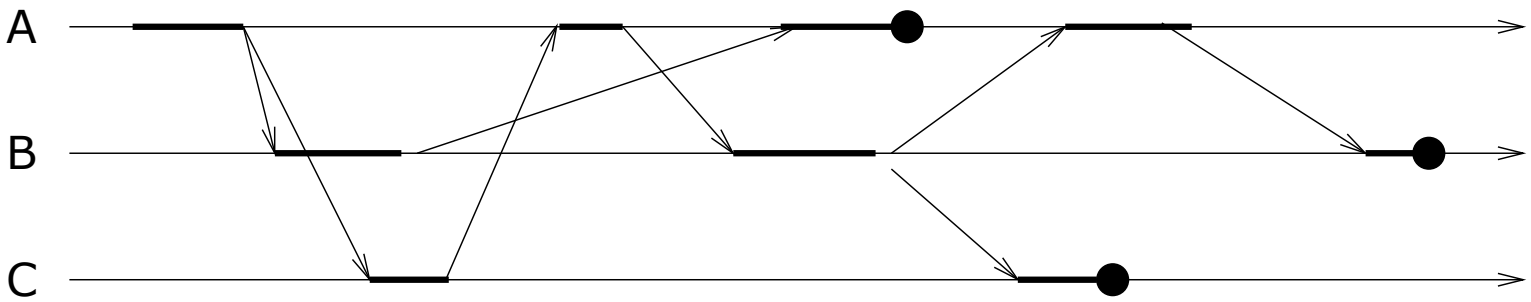
- Sûreté : Pas de fausse détection :
  $Term \Rightarrow (\forall\ i\ ::\ P_i.passif \wedge EnTransit = \emptyset)$
- Vivacité : La terminaison finit par être détectée :
  $(\forall\ i\ ::\ P_i.passif \wedge EnTransit = \emptyset) \rightsquigarrow Term$



Définition d'un calcul diffusant

- Un processus initial émet un ou plusieurs messages
- Puis, tous les processus adoptent le même comportement :

```
loop {   /* un pas de calcul  */
         recevoir(m) ;
         traiter m ;
         envoyer 0 à N − 1 messages ;
}
```



$\Rightarrow$ Les phases actives peuvent être vues comme atomiques

Terminaison sur un anneau (Misra, 1983)

Principe

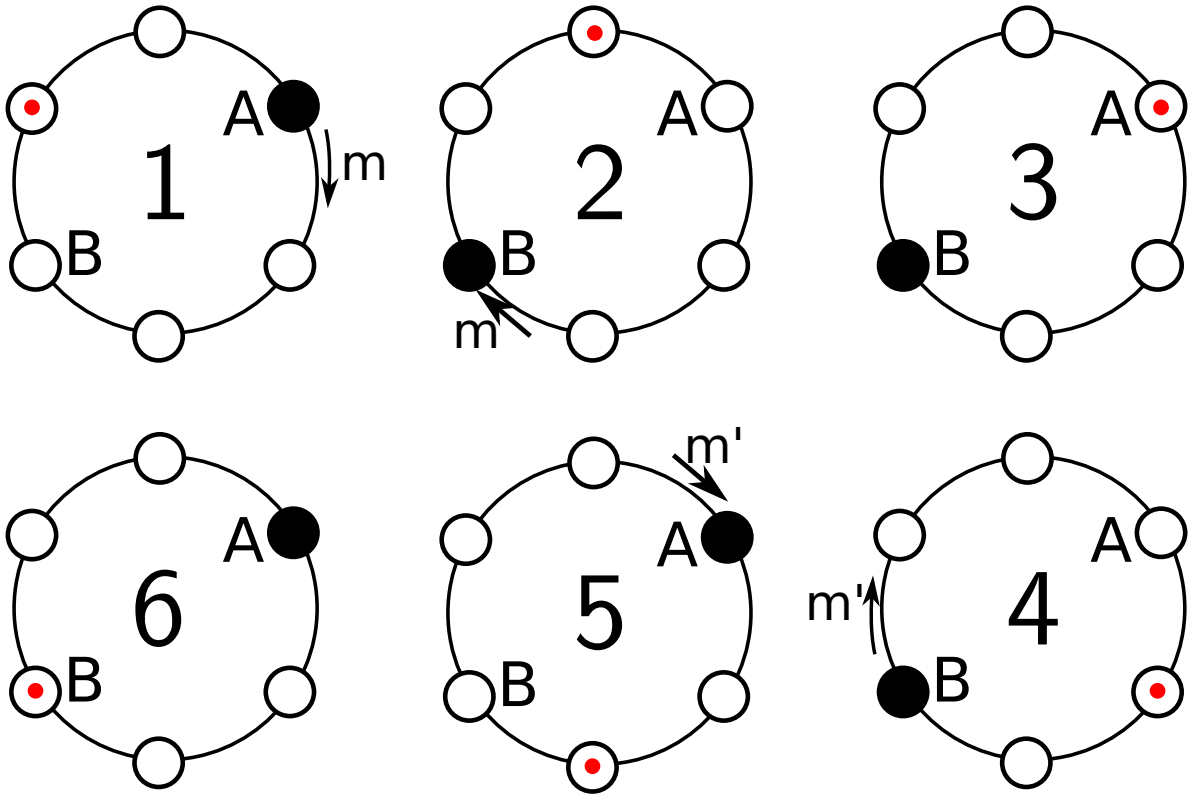
Les sites sont organisés en anneau (communication FIFO depuis le précédent / vers le suivant mais à destinataire arbitraire).
Parcourir l'anneau et vérifier que tous les sites sont passifs.

Difficulté

Un message émis avant le passage du visiteur sur le site émetteur peut être reçu après le passage du visiteur sur le site récepteur et réactiver un site trouvé passif

$\Rightarrow$ faire deux tours en vérifiant qu'aucun site n'a changé d'état entre temps

Terminaison – Misra



- tous trouvés passifs, mais calcul pas terminé.
- A et B trouvés passifs, mais ils ont été actifs entre temps.

$\Rightarrow$ faire deux tours en vérifiant qu'aucun site n'a changé d'état entre temps

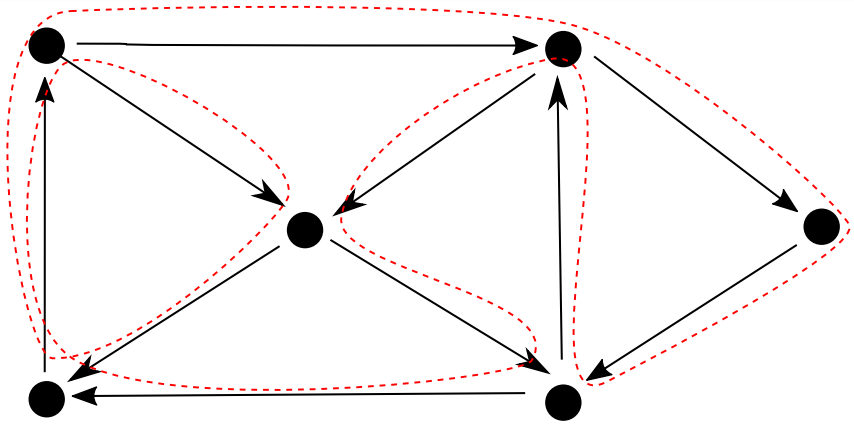
Terminaison sur un anneau

```
Process P(i : 0..N-1) {
 variables : couleur ∈ {blanc,noir}
             état ∈ {actif,passif}
             jeton ∈ {true,false}
 ⋮
  on reception message_applicatif :    // action
       couleur ← noir;
       état ← actif;
  on reception jeton(val) :            // réception jeton
       jeton ← true; nb ← val;
       if (nb = N ∧ couleur = blanc) then TERMINAISON DÉTECTÉE
  on jeton ∧ état = passif :           // envoi jeton
       if (couleur = blanc) then send jeton(nb + 1) to P_{i⊕1}
                            else send jeton(1) to P_{i⊕1}
       couleur ← blanc;
       jeton ← false;
  else :                               // attente
       état ← passif;
```

Terminaison avec un anneau logique

Graphe de communication arbitraire

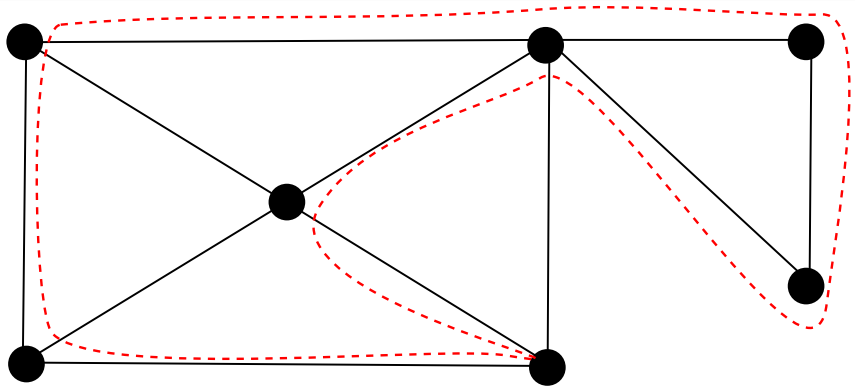
- Circuit logique contenant tous les arcs (éventuellement plusieurs fois)
- Communication FIFO : le jeton ne peut pas dépasser un message antérieur sur le même arc
- Terminaison comme précédemment, avec N = longueur du circuit



Terminaison avec un anneau logique

Graphe de communication arbitraire – avec la causalité

- Circuit logique contenant tous les sites (mais pas nécessairement tous les arcs)
- Communication causale : le jeton en transit depuis un site s ne peut pas arriver sur s' avant les messages émis avant sa visite sur s et envoyés directement de s vers s'
- Terminaison comme précédemment, avec $N =$ longueur du circuit



Plan

1. UN paramètre numérique
   - Élection
   - Exclusion mutuelle
   - Détection de la terminaison
2. DEUX paramètres numériques
   - Le problème du consensus
   - Système synchrone
   - Système asynchrone
3. TROIS paramètres numériques
4. Paramètres non numériques
   - Modèles
   - Détecteur de fautes

Le consensus

Définition

Soit un ensemble de processus $p_1, \ldots, p_n$ reliés par des canaux de communication.
Initialement : chaque processus p_i propose une valeur v_i.
À la terminaison de l'algorithme : chaque processus p_i décide d'une valeur d_i.
Propriétés :

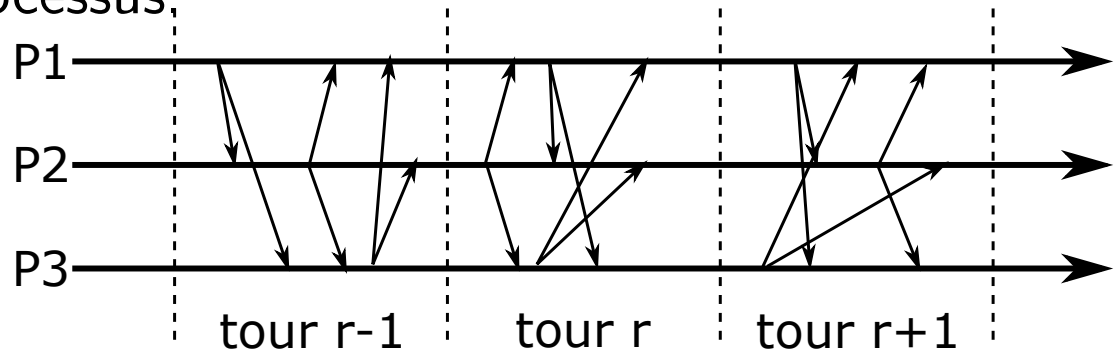
- Accord : la valeur décidée est la même pour tous les processus corrects
- Intégrité : tout processus décide au plus une fois (sa décision est définitive)
- Validité : la valeur décidée est l'une des valeurs proposées
- Terminaison : tout processus correct décide au bout d'un temps fini

Modèle temporel

Synchrone

borne supérieure connue sur le temps de transmission et sur l'avancement des processus.

Usuellement, algorithmes fonctionnant par tours, synchronisés sur tous les processus.



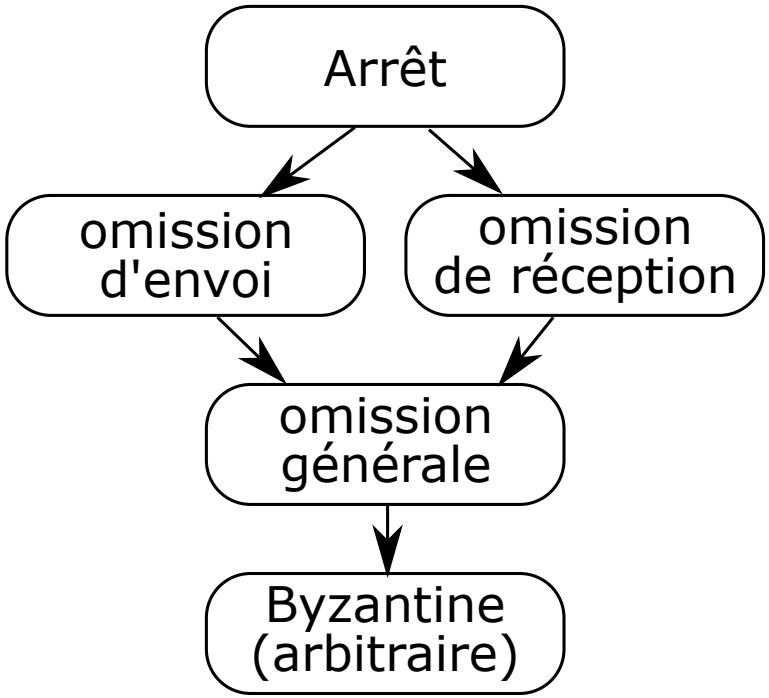
Asynchrone

Pas de borne connue : avancement arbitrairement lent des processus et du réseau.

Modèle moins contraint, plus réaliste (mais plus difficile)

Défaillances d'un processus

- Arrêt (*crash failure* ou panne franche) : le processus fonctionne correctement jusqu'à un point où il cesse définitivement d'agir.
- Omission
  - omission en émission : le processus omet certaines émissions qu'il aurait dû faire, ou cesse définitivement.
  - omission en réception : le processus ignore certains messages en réception, ou cesse définitivement.
- Arbitraire (*byzantine failure*) : le processus ment (par omission ou par contenu arbitraire des messages envoyés)



Réalisabilité du consensus

défaillance	synchrone	asynchrone
non	faisable	faisable
arrêt	faisable f pannes $< n$ processus $\Omega(f+1)$ tours	impossible (FLP 1985)
omission	faisable f pannes $< n/2$ processus	impossible
byzantine	faisable $f \leq \lfloor (n-1)/3 \rfloor$ processus $\Omega(f+1)$ tours	impossible

Synchrone, défaillance d'arrêt

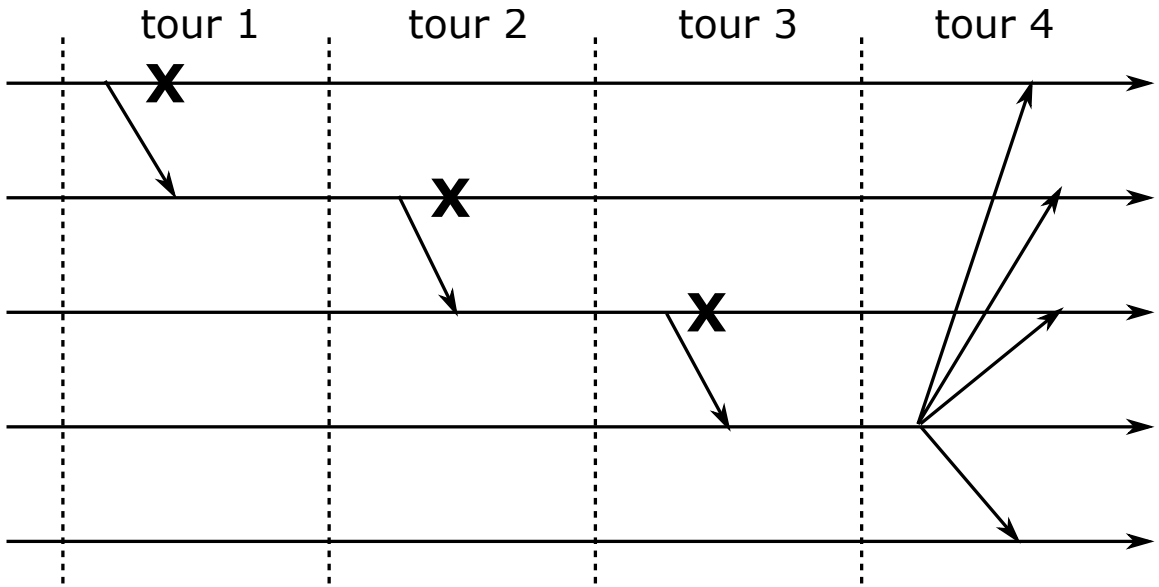
Tolérance à f défaillances ($f < n$).

Principe

Au i-ième tour, le processus i diffuse sa valeur. Après $f+1$ tours, on est sûr qu'au moins un processus correct a diffusé une valeur reçue par au moins $n-f$ processus corrects.

```
Processus P_i(v_i),  0 ≤ i < n
local x
on start :
    x ← v_i
on réception(v) :
    x ← v
on time i × Δ, i ≤ f :     // au i-ième tour pour P_i
    envoyer(x) à tous les autres
on time (f + 1) × Δ :
    décider x
```

Synchrone, défaillance d'arrêt



- $f + 1$ tours
- $f + 1$ diffusions
- décision simultanée
- non équitable : les valeurs des processus $f + 1, \ldots, n$ ne sont pas considérées

Algorithme équitable à $f+1$ tours

Principe

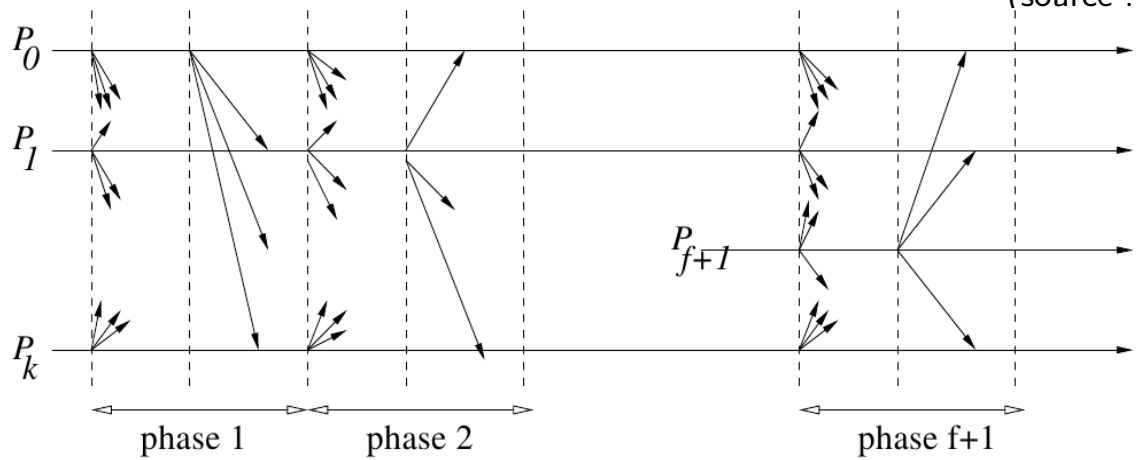
À chaque tour, chaque processus diffuse la plus petite valeur qu'il connaît (uniquement si elle a changé).

```
Processus P_i(v_i), 0 ≤ i < n
local x, prevx, received
on start :
    x ← v_i, prevx ← ⊥
on réception(v) :
    received ← received ∪ {v}
on time k × Δ, 0 ≤ k ≤ f :        // à chaque tour
    prevx ← x
    x ← min(received ∪ {x})  // sélection déterministe
    received ← ∅
    si x ≠ prevx alors diffuser(x)
on time (f + 1) × Δ:
    décider x
```

Synchrone, défaillance byzantine (*Phase King*)

- $f + 1$ phases, chaque phase a un unique roi, fixé a priori
- Chaque phase contient deux tours :

(source : Kshemkalyani)



  - Tour 1 : chaque site diffuse son estimation à tous.
    Chaque site reçoit les valeurs proposées puis détermine si une valeur est proposée par une majorité qualifiée ($> n/2 + f$), ou une majorité simple ($> n/2$)
  - Tour 2 : le roi fixe son estimation à sa valeur majoritaire (sa propre valeur si pas de majorité) et la diffuse.
    Chaque site fixe sa nouvelle estimation à la valeur reçue en tour 1 avec majorité qualifiée, ou sinon à la valeur reçue du roi.

Algorithme avec “roi de phase” – justification

$f+1$ phases, $(f+1)(n+1)(n-1)$ messages,
$f < \lceil n/4 \rceil$ défaillances

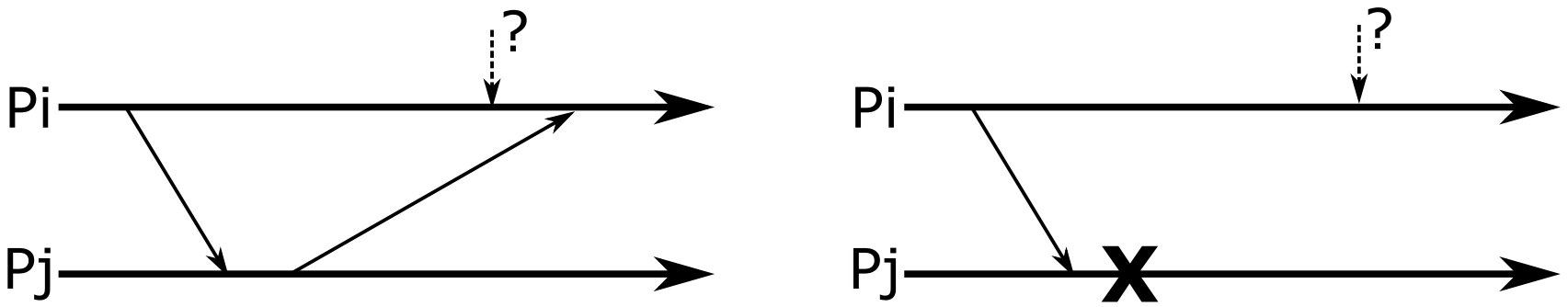
- Parmi les $f+1$ phases, au moins une où le roi est correct
- Dans cette phase, les processus corrects obtiennent la même estimation que le roi : soit p_i et p_j corrects :
  - ou p_i et p_j ont chacun une majorité qualifiée ($> n/2+f$) et le roi a alors aussi cette même estimation
  - ou p_i a une majorité qualifiée ($> n/2+f$) et p_j utilise la valeur du roi ($> n/2$)
  - ou p_i et p_j utilisent l'estimation du roi
- Si tous les processus corrects ont la même estimation au début d'une phase, ils garderont cette même valeur à la fin (même si le roi est byzantin).

Impossibilité du consensus en asynchrone avec arrêt

Résultat d'impossibilité (Fischer, Lynch, Paterson, 1985)

Le consensus est impossible à réaliser dans un système asynchrone où un seul processus peut subir une défaillance d'arrêt.

Intuitivement, il est impossible de distinguer un processus lent d'un processus arrêté.



Plan

1. UN paramètre numérique
   - Élection
   - Exclusion mutuelle
   - Détection de la terminaison
2. DEUX paramètres numériques
   - Le problème du consensus
   - Système synchrone
   - Système asynchrone
3. TROIS paramètres numériques
4. Paramètres non numériques
   - Modèles
   - Détecteur de fautes

k consensus

- k Accord : au plus k valeurs distinctes sont décidées pour l'ensemble des processus corrects (Consensus basique : $k = 1$)

Le k-consensus est faisable en asynchrone/arrêt avec $f < k < n$

(f = nombre max de fautes, k = nb max de valeurs distinctes, n = nombre de processus)

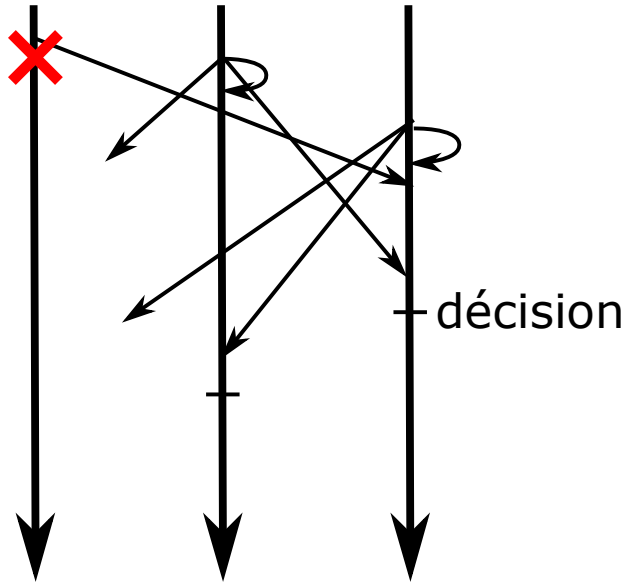
k-consensus en synchrone

```
Processus P_i(v_i), 0 ≤ i < n
local x, received
on start :
    x ← v_i
on réception(v) :
    received ← received ∪ {v}
on time r × Δ, 0 ≤ r ≤ ⌊f/k⌋ :        // à chaque tour
    x ← min(received ∪ {x})  // sélection déterministe
    diffuser(x)
    received ← ∅
on time (⌊f/k⌋ + 1) × Δ :
    décider min(received ∪ {x})
```

S'il y a eu y défaillances à la fin du round r, il y a au plus $y + 1$ valeurs distinctes à la fin de r.

k-consensus en asynchrone

- Chaque processus envoie sa valeur à tous les autres
- Quand un processus a reçu (au moins) $n - (k - 1)$ valeurs, il décide le min.



Si $f < k < n$ alors :

- Terminaison en temps fini (non borné) pour les processus corrects
- Au plus f valeurs non reçues par tous $\Rightarrow$ au plus $f + 1$ décisions distinctes

Plan

1. UN paramètre numérique
   - Élection
   - Exclusion mutuelle
   - Détection de la terminaison
2. DEUX paramètres numériques
   - Le problème du consensus
   - Système synchrone
   - Système asynchrone
3. TROIS paramètres numériques
4. Paramètres non numériques
   - Modèles
   - Détecteur de fautes

Paramètres non numériques

- Ordre de délivrance : fifo, causal, etc (cf suite)
- Structure du réseau : anneau, maillé...
- Défaillances : perte de message, arrêt de processus...

Nombre de cas fini $\Rightarrow$ exploration exhaustive mais bourrin.

Peut-on réutiliser le résultat de la vérification effectuée avec un modèle, pour conclure pour plusieurs ?

Détecteur de fautes : Motivation

Résultat d'impossibilité (Fischer, Lynch, Paterson, 1985)

Le consensus est impossible à réaliser dans un système asynchrone où un seul processus peut subir une défaillance d'arrêt.

- Le consensus en asynchrone est impossible car on ne sait pas distinguer un processus défaillant (arrêté) d'un processus lent.
- On va supposer l'existence d'un détecteur de fautes, qui indique si un processus est fautif ou non.
- FLP $\Rightarrow$ un détecteur parfait est impossible.
- Il existe des détecteurs imparfaits (i.e. qui peuvent se tromper) qui suffisent pour réaliser le consensus !
- FLP $\Rightarrow$ ces détecteurs imparfaits sont impossibles mais on peut en construire des approximations réalistes.

Détecteur de fautes

Définition (Chandra, Toueg, 1991)

- Un détecteur de fautes est un service réparti composé de détecteurs locaux à chaque processus (site).
- Un détecteur fournit à son processus local une liste des processus qu'il suspecte d'être défaillants.
- Les détecteurs locaux coopèrent (ou pas) pour établir cette liste.

- Propriétés :
  - Complétude : peut-on ne pas suspecter un processus défaillant ?
  - Exactitude : peut-on suspecter un processus correct ?
- Équivalent à un oracle (éventuellement imparfait)

Complétude

Complétude (*completeness*)

- Complétude forte : tout processus défaillant finit par être suspecté par tout processus correct
- Complétude faible : tout processus défaillant finit par être suspecté par un processus correct

Complétude faible et complétude forte sont équivalentes :

- En complétude faible, tout processus défaillant finit par être détecté par au moins un processus
- Périodiquement, chaque processus diffuse sa liste de processus suspectés
- Alors tous les processus finiront par obtenir l'information de suspicion = complétude forte
- (hypothèse : la diffusion est fiable, ce qui est réalisable en asynchrone avec défaillance d'arrêt)

Exactitude

Exactitude permanente (*accuracy*)

- Exactitude forte : aucun processus correct n'est jamais suspecté par aucun autre processus correct
- Exactitude faible : il existe un processus correct qui n'est jamais suspecté par aucun autre processus correct

Exactitude inévitable

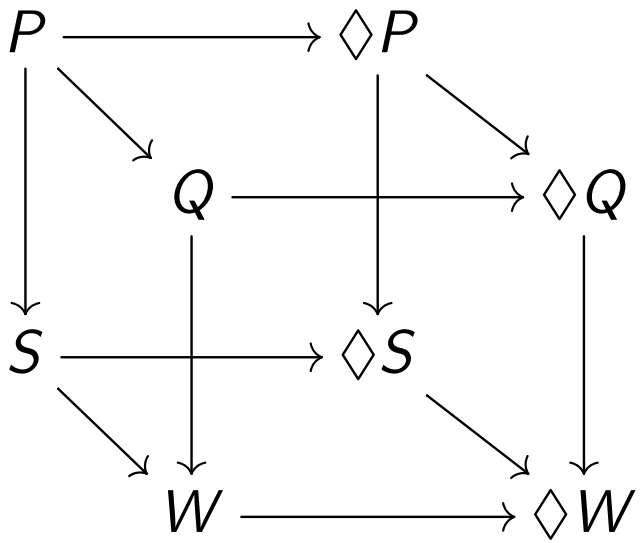
- Exactitude finalement forte : au bout d'un certain temps, aucun processus correct n'est plus jamais suspecté par aucun autre processus correct
- Exactitude finalement faible : au bout d'un certain temps, il existe un processus correct qui n'est plus jamais suspecté par aucun autre processus correct

Finalement = inévitablement = *eventually*

Classes de détecteurs de fautes

	Exactitude			
	forte	faible	finalement forte	finalement faible
Complétude forte	P	S	$\Diamond P$	$\Diamond S$
Complétude faible	Q	W	$\Diamond Q$	$\Diamond W$

P = perfect, S = strong, W = weak



Synthèse

détecteur	nombre de fautes	nombre de tours
P	$n-1$	$f+1$
S	$n-1$	n
$\Diamond S$, $\Diamond W$	$\frac{n-1}{2}$	fini non borné

Plus faible détecteur de fautes

Le détecteur de fautes $\Diamond W$ (complétude faible, exactitude finalement faible) est le plus faible détecteur de fautes permettant de résoudre le consensus en asynchrone avec défaillance d'arrêt.