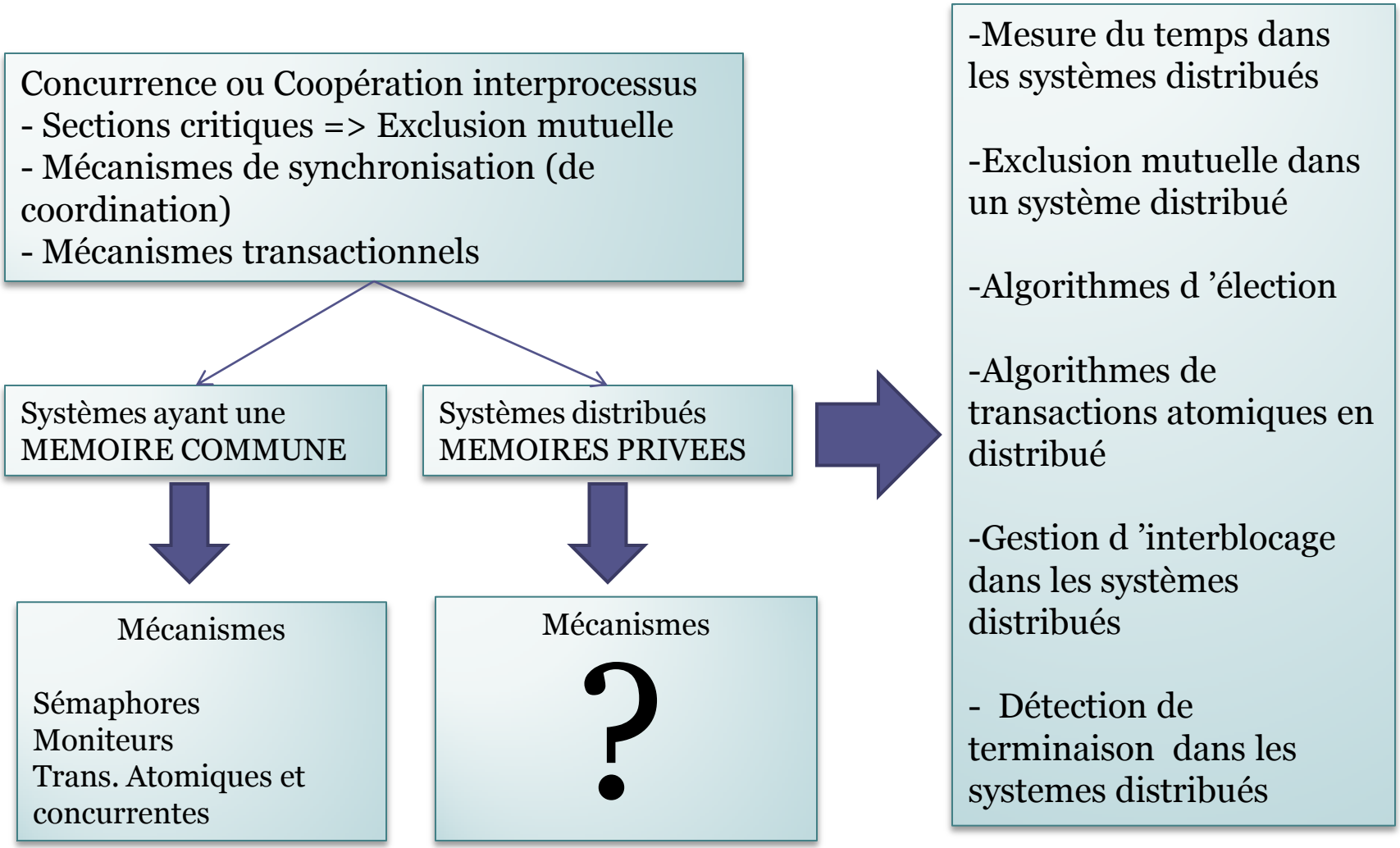
Synchronisation dans les systèmes distribués

Chapitre 3

Plan

- Synchronisation d'horloge
- Exclusion-mutuelle
- Algorithmes d'élection
- Transactions atomiques
- Détection de terminaison

Introduction



Synchronisation d'horloge

Introduction

Propriétés d'algorithmes distribués

1- les informations importantes sont disséminées sur plusieurs Machines

2- les processus prennent des décisions fondées uniquement sur l'information disponible Localement

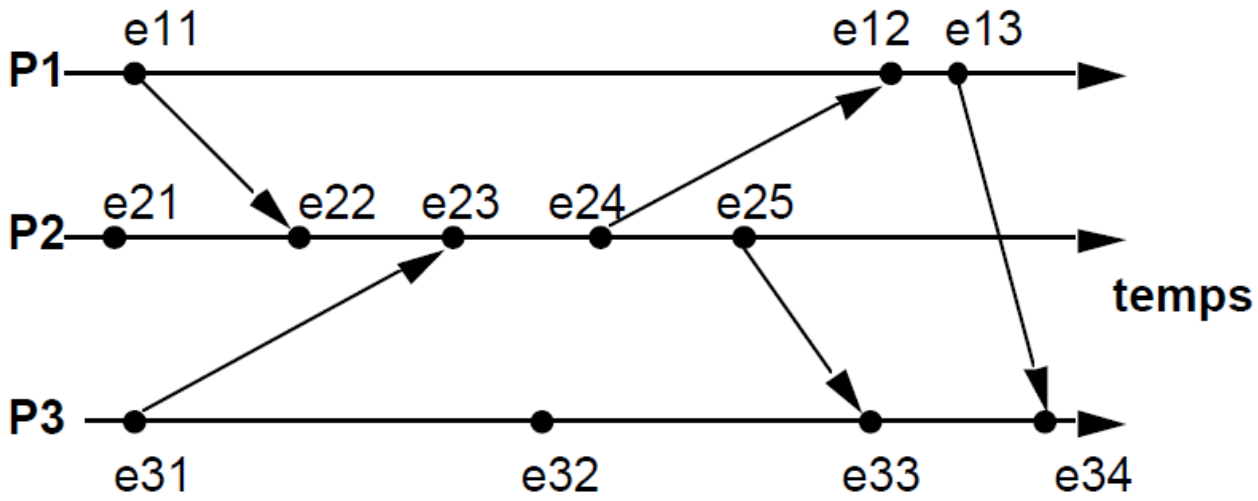
3- la garantie de la fiabilité doit être renforcée

4- il n 'existe pas d 'horloge commune ni de source de temps universel

Synchronisation de l'Horloge :But

- Synchronisation de l'Horloge dans un système distribué a pour but :
  - Ordonnancement temporel d'événement qui se produisent dans des processus concurrent
  - Synchronisation entre l'émetteur et le récepteur d'un message
  - Synchronisation entre plusieurs processus pour le déclanchement d'une action commune
  - Sérialisation des accès concurrent sur une ressource partagée

Un modèle pour l'étude de la synchronisation dans un système réparti



Système *= ensemble de processus (1 par site) + canaux de communication*
Processus *= séquence d'événements locaux + mémoire locale + horloge locale*
Evénement *= changement d'état du processus (événement interne), ou émission d'un message, ou réception d'un message*

Calculs et observations distribués

- *Observation d'un calcul réparti E = "linéarisation" de E (définition d'un ordre total, sur les événements de E)*

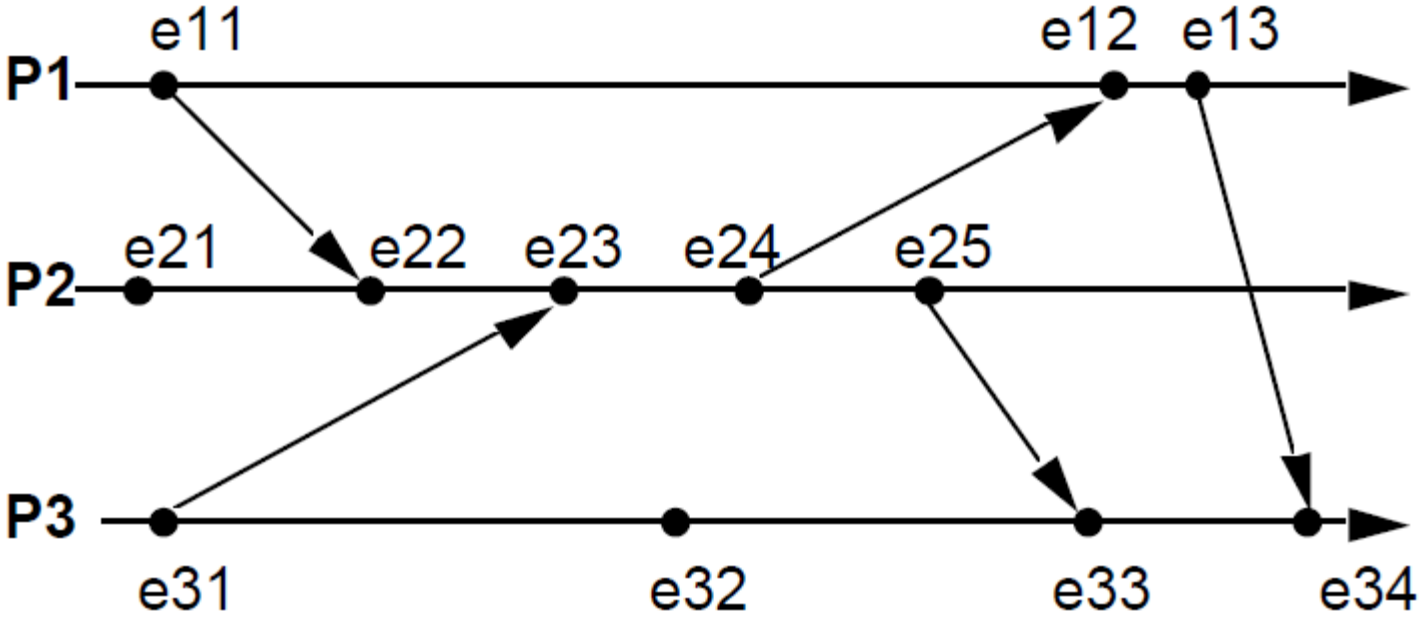
- *Exemple : vue de la séquence des événements de E par un observateur interne ou externe au système*

- *Observation valide = compatible avec la relation de précédence causale (pourrait être observée par un observateur externe réel)*

Calculs et observations distribués

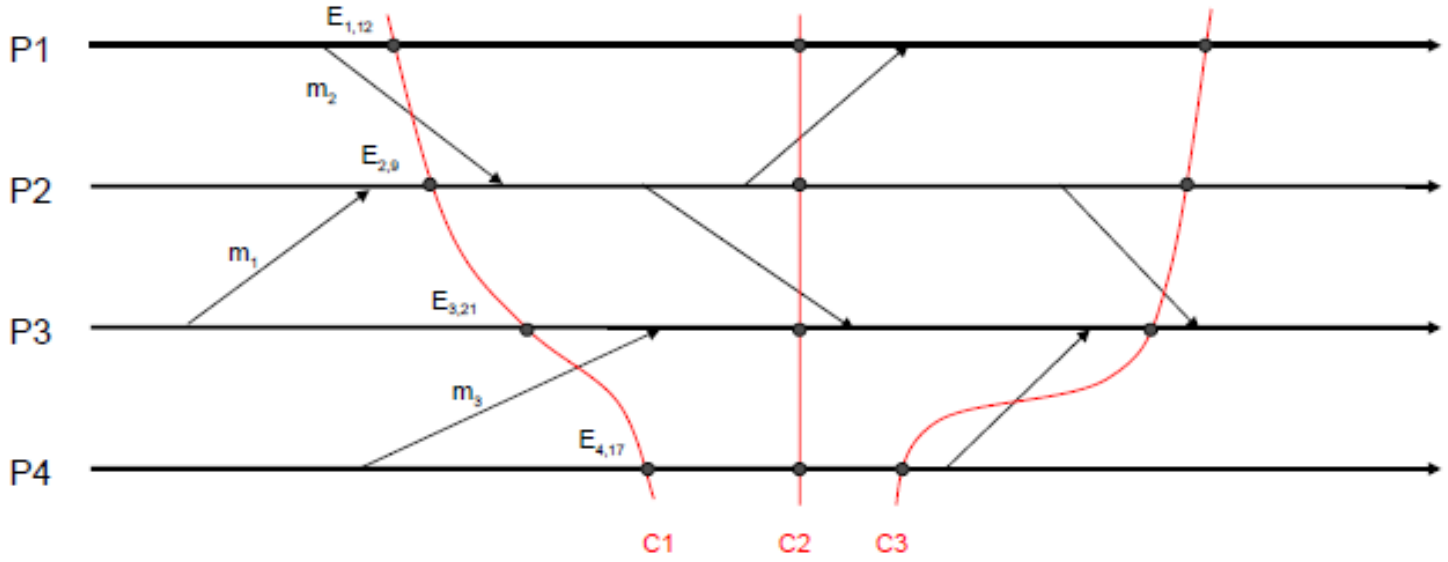
- ***Exemple :***

- *e11 e21 e31 e22 e23 e32 e24 e25 e12 e33 e13 e34 valide*
- *e11 e21 e31 e22 e32 e23 e24 e25 e12 e33 e13 e34 Valide*
- *e11 e21 e31 e22 e23 e32 e24 e25 e12 e33 e34 e13 invalide*



Etat global

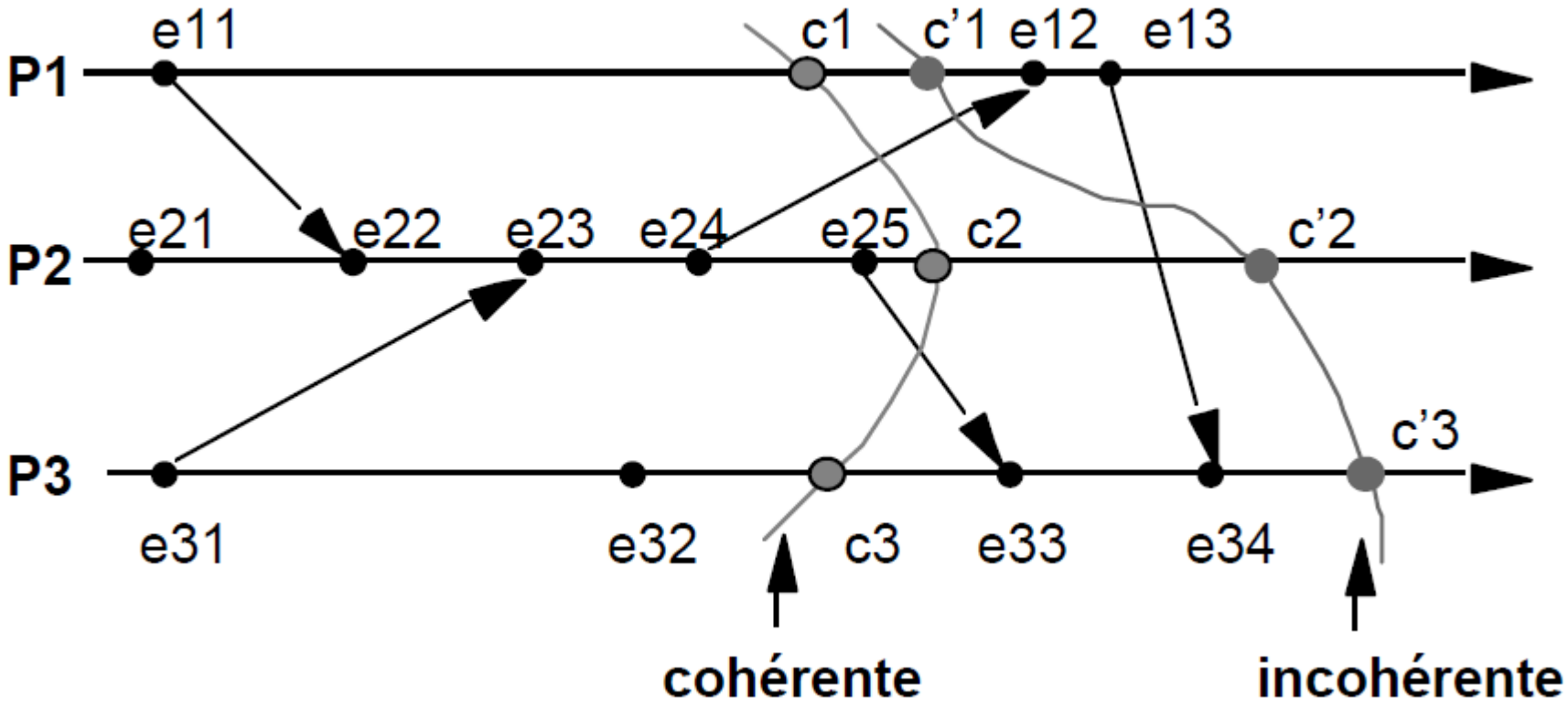
Un état global peut être représenté par une courbe, appelée aussi **coupure**, sur le diagramme temporel du système réparti. La coupure divise le diagramme en deux zones (sur chaque processus) : passé et futur.



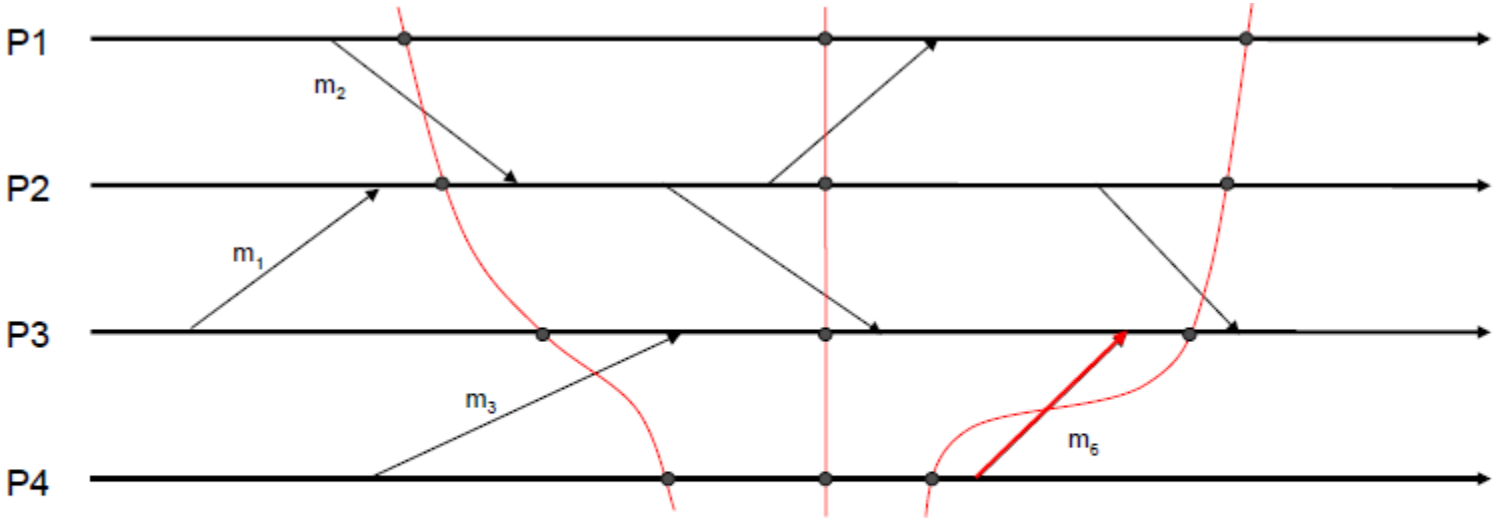
$E_{C1}(SR) = E(P_1) = E_{1,12} \cup E(P_2) = E_{2,9} \cup E(P_3) = E_{3,21} \cup E(P_4) = E_{4,17} \cup E(C_{12}) = \{m_2\} \cup E(C_{43}) = \{m_3\}$

Coupures cohérentes

- ***Cohérence = respect de la causalité (un message ne peut pas venir du futur)***



Coupures cohérentes



C1 et C2 sont cohérentes

C3 **n'est pas** cohérente :
réception(m_6) $\in$ Passé(C3) et
émission(m_6) $\notin$ Passé(C3)

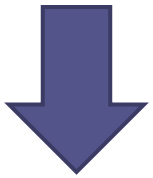
Temps Physique vs Temps Logique

- Temps logique : Ordre des évènements
- Temps physique : Quant s'est déroulé un évènement

Temps global logique

Horloge logique

Temporisateur associé à chaque processeur permettant de générer un compteur de temps local (des impulsions d ’horloge)



Problème de dérive d ’horloge

Dans un système distribué au fil du temps, la synchronisation des horloges se dégrade

Introduction des erreurs

Horloge Logique

- Objectif : ordonner des événements dans un système distribué
- Chaque processeur maintient une horloge locale
- Pas de point central pour fournir le temps
- Ne donne pas le temps d'un événement, elle s'intéresse à préciser les événements qui se sont déroulés avant ou après

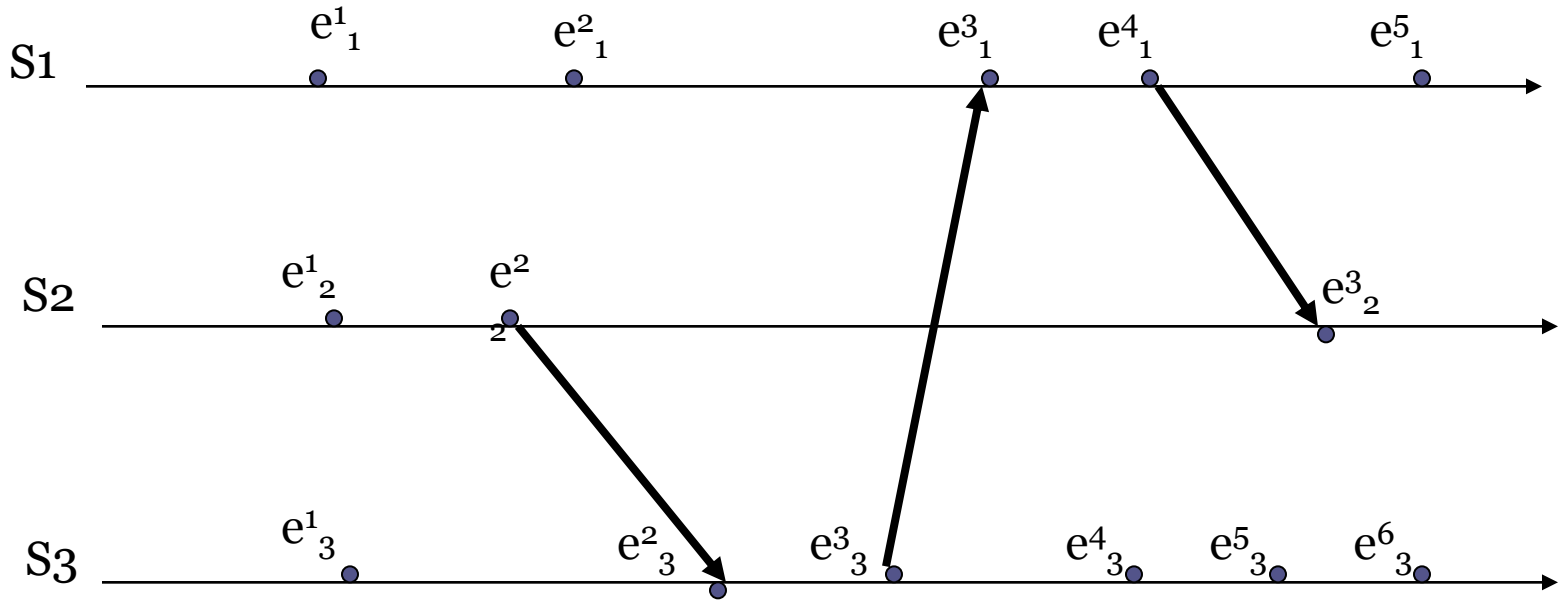
Dans un système réparti composé de n sites connectés par des canaux :

- **Les événements sur un même site sont totalement ordonnés (émission, réception, interne)**
- **Pour chaque message, l'événement émission précède toujours sa réception.**

La fermeture transitive de ces relations de précédence définit une relation de dépendance causale notée $\rightarrow$ sur l'ensemble des événements produits par une exécution répartie.

Cette relation est un ordre partiel.

Exemple : diagramme d'une exécution répartie.



$e^1_1 \rightarrow e^2_1$

$e^1_2 \rightarrow e^3_1$

Deux événements x et y tels que ni $x \rightarrow y$ ni $y \rightarrow x$ sont dits Indépendants ou concurrents. On notera $x \mid\mid y$.

Problème :

Trouver des mécanismes qui permettent d'associer des dates aux événements concernés tels que si :

a $\rightarrow$ ***b*** **alors la date associée à *b* doit être relativement à un temps logique global, après la date associée à *a*.**
(propriété de monotonie)

Mécanismes d'estampillage

- **Deux mécanismes d'estampillage essentiels :**
  - Temps linéaire (Lamport-78)
    - Temps logique représenté par un entier
  - Temps vectoriel (Mattern-89, Fidge-89)
    - Temps représenté par un vecteur de dimension n

- **Modèle**

Ces deux mécanismes obéissent au même modèle :

- **Structure de données pour représenter le temps**

A chaque site sont associées des variables qui lui permettent :

- **Mesurer sa propre progression, ce qui est assuré par son horloge logique locale (mise à jour par la règle1)**

- **Avoir une bonne représentation du temps logique global; c'est une vue locale du temps global (mise à jour par la règle2).**

➢ Protocole

Assure que l'horloge logique locale et la vue locale du temps Global de chaque site sont gérées de manière cohérente relativement à la relation $\rightarrow$.

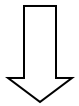
- Règle 1:

Avant de produire un événement (émission, réception, interne), un site doit incrémenter son horloge locale (parce qu'il progresse).

- Règle 2:

Pour que la date (estampille) correspondant à la réception d'un message soit après la date correspondant à l'événement émission du même message, $\Rightarrow$ chaque message m

transporte la valeur du temps logique global vu par l'émetteur Au moment de l'émission.



Ceci permet au récepteur de mettre à jour sa vue du temps global. Puis il exécute la règle 1.

Temps linéaire

❑ **Protocole**

A chaque site est associée une variable entière h_i ayant des valeurs croissantes.

Règle 1: **Avant de produire un événement (émission, réception, interne) Faire :**

$$h_i := h_i + d \ (d > 0)$$

% Chaque fois que cette règle est exécutée, d peut avoir une valeur
% différente.

Règle 2: **Lorsqu'un site S_i reçoit un message *(m,h)* Faire :**

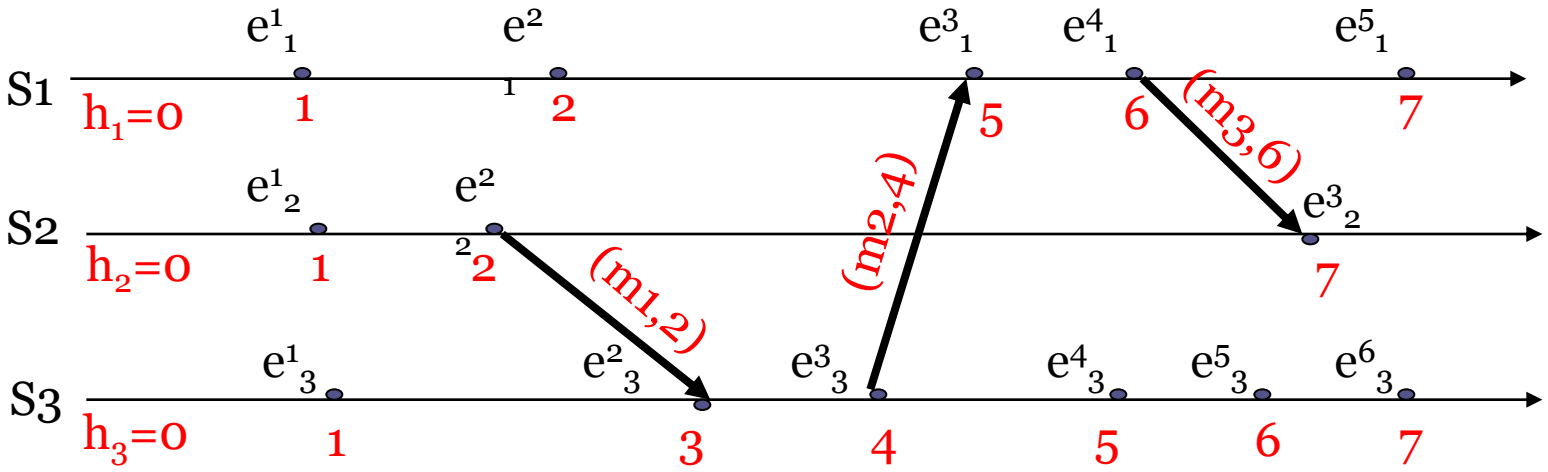
$h_i := max(h_i, h)$
Puis exécuter la règle 1.

❖ Propriétés

Il est possible d'utiliser ce mécanisme d'estampillage pour construire un ordre total noté $\rightarrow^{T}$.

- **Estampille (e) ≡ date d'occurrence + identité du site qui a produit e.**
- **Si on a deux événements x et y estampillés respectivement par *(h,i)* et *(k,j),* on définit :**

$$x \rightarrow^{T} y \Leftrightarrow (h < k \text{ ou } (h = k \text{ et } i < j))$$



- Les estampilles ne sont pas “denses”

  si H(e) < H(e'), on ne peut pas savoir s’il existe e"

  tel que e précède e" et/ ou e"‘ précède e ’.

  On ne peut pas prendre immédiatement une décision

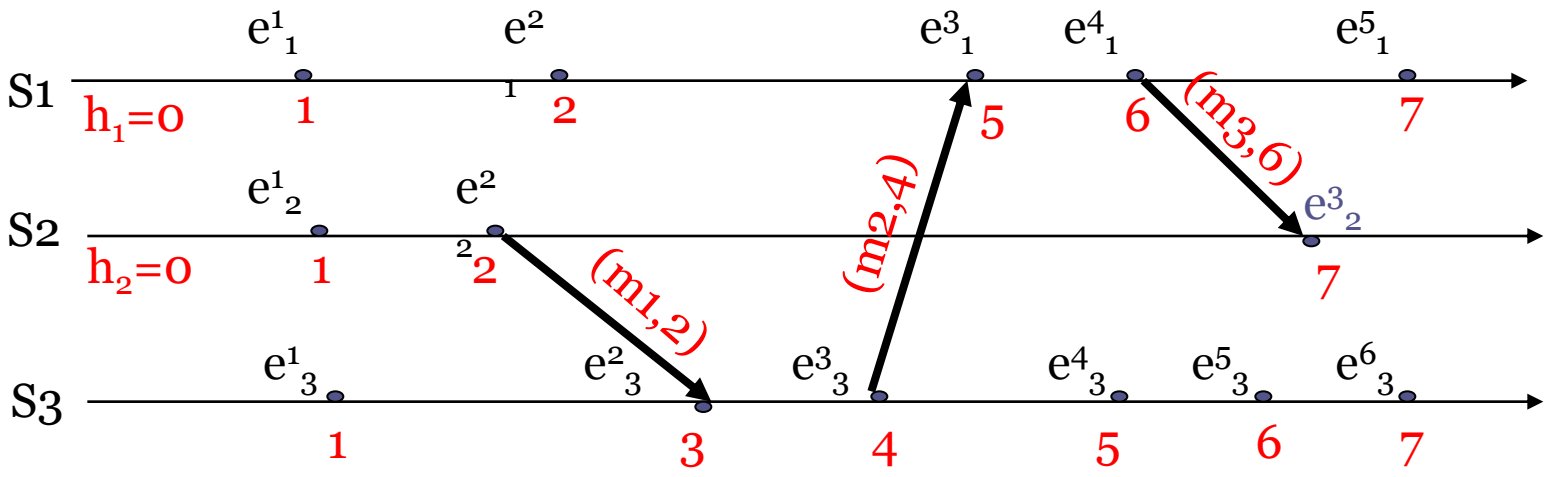
- Pour certaines applications (mise au point, mesure du parallélisme) on a besoin de caractériser l’indépendance causale.

  Propriété recherchée pour une horloge V:

  e $\rightarrow$ e' est équivalent à V(e) < V(e')

Horloges vectorielles

- **Une méthode de datation causale : les historiques**
  - Rappel : passé d'un événement e
  - hist(e) = l' ensemble des e' tels que e' → e



Hist(e32) = {e11,e21,e31,e41,e12,e22,e32,e13,e23,e33}

- Idée : utiliser le passé d'un événement e pour la datation
- Le passé permet de déterminer la dépendance causale :
  - hist(e) : ensemble des événements e' tels que e' $\rightarrow$ e

    e $\rightarrow$ e' alors e $\in$ hist(e')

    e || e' alors (e $\notin$ hist(e')) et (e' $\notin$ hist(e))
- Inconvénient : taille de hist(e)
- Remède : pour définir hist(e), un événement par site suffit.

  C'est l'idée des horloges vectorielles.

- Le temps est ici représenté par un vecteur de dimension n.

❑ Protocole

Chaque site S_i est doté d'un vecteur $vt_i[1..n]$ où :

- $vt_i[i]$; **décrit l'évolution du temps logique du site S_i : l'horloge logique locale de S_i. Elle prend des valeurs croissantes générées localement.**
- $vt_i[j]$: **Représente la connaissance qu'a le site S_i sur l'évolution du temps sur le site S_j. C'est une image locale de la valeur $Vt_j[j]$.**
- **Le vecteur vt_i constitue une vue locale du temps logique global utilisé pour estampiller les événements.**

Règle 1:

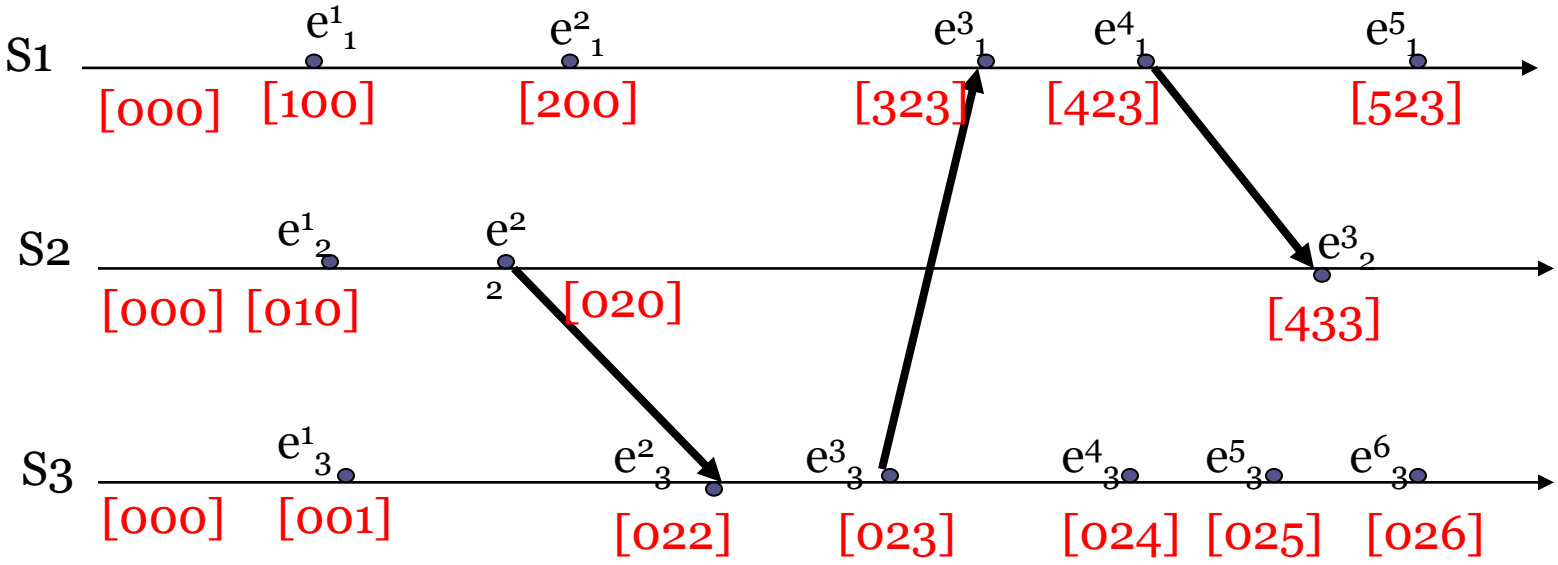
Avant de produire un événement
Faire :

$$Vt_i[i] := vt_i[i] + d \quad (d > 0)$$

Règle 2:

A la réception de *(m,vh)*
Faire :

Pour k de 1 à n faire vti[k] := max(vti[k],vh[k])
Puis exécuter la règle 1.



- Propriétés

  - $vh \leq vk \Leftrightarrow \forall x : vh[x] \leq vk[x]$
  - $vh < vk \Leftrightarrow vh \leq vk$ et $\exists x : vh[x] < vk[x]$
  - $vh \,||\, vk \Leftrightarrow \neg(vh < vk)$ et $\neg(vk < vh)$

  - Si on a deux événements x et y estampillés respectivement vh et vk alors :

    $x \rightarrow y \Leftrightarrow vh < vk$
    $x \,||\, y \Leftrightarrow vh \,||\, vk$

Horloges physiques

- Algorithme de Lamport propose une solution pour ordonner les évènements de façon non ambiguë
- Mais horloges logiques associées aux évènements ne correspondent pas forcément à la date réelle à laquelle ils se produisent
- Dans certains systèmes (trafic aérien, commerce, transactions) , le temps réel est important : un certain nombre d'algorithmes permettant de gérer ce problème existent

Propriétés des horloges physiques

Déviation, ou écart (skew) au temps t : différence entre deux horloges (par ex. sur deux machines d'un réseau)

Précision : déviation par rapport au temps réel (universel)

Dérive (drift) : divergence (déviation croissante) entre horloges due à des fréquences différentes (notamment divergence entre une horloge et l ' horloge“ idéale ” donnant l ' heure exacte)

Taux de dérive (drift rate) : mesure de la dérive entre deux horloges (en déviation par seconde).

Qu' est-ce que le “temps réel” ?

Une convention internationale définit le temps réel

- UTC : Coordinated Universal Time
- Utilise une horloge atomique (taux de dérive 10^{-13} s/s) ; ajustement périodique par rapport au temps astronomique
- L'heure universelle est diffusée par radio et satellites (GPS)
- Un ordinateur muni d'un récepteur peut donc synchroniser périodiquement son horloge interne, et fonctionner comme serveur de temps sur un réseau local

Synchronisation d'horloges physiques

Synchronisation externe

À partir d'une source de temps S (serveur), un ensemble d'horloges Hi est synchronisé de telle sorte que dans tout intervalle Δt de temps réel :

$|S(t) - H_i(t)| < D$ (écart borné) pour tout $t \in \Delta t$

Synchronisation interne

Les horloges Hi sont mutuellement synchronisées, de sorte que dans tout intervalle Δt de temps réel :

$| H_i(t)| - H_j(t)| < D$ pour tout couple i, j et pour tout $t \in \Delta t$

Synchronisation interne n' implique pas synchronisation externe : l'ensemble des horloges peut collectivement dériver par rapport à un serveur

Synchronisation externe : algorithme de Cristian (1)

Problème : synchroniser un ensemble d'horloges à partir d'un serveur de temps (lui-même alimenté par le temps universel UTC)

Difficulté : tenir compte du temps de transfert, a priori inconnu

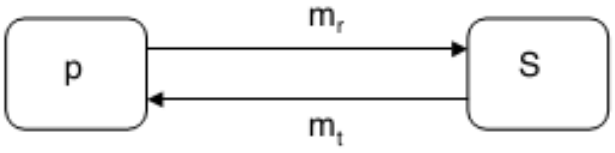
Solution : mesurer le temps d'aller-retour

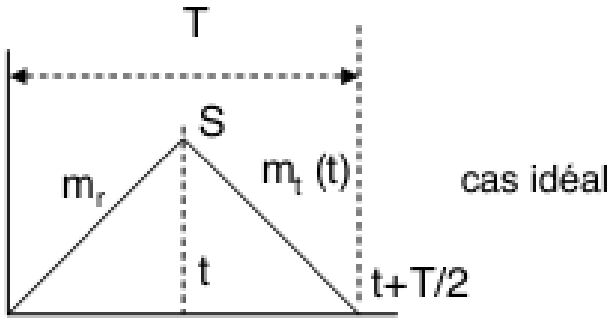
Synchronisation externe : algorithme de Cristian (1)

p demande l'heure à S (message m_r) et reçoit t (message m_t)

Par ailleurs p a mesuré le temps T d' aller-retour de p à S

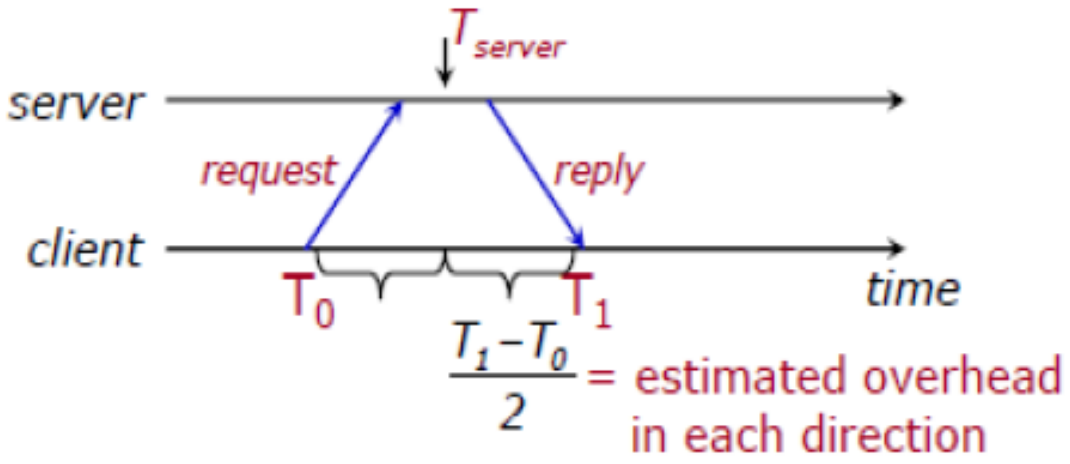
p règle son horloge au temps t + T/2





Synchronisation externe : algorithme de Cristian (2)

Chaque machine interroge Périodiquement le serveur de temps pour avoir l ’heure courante afin de pouvoir corriger d ’une manière adéquate son horloge



Client sets time to: $T_{new} = T_{server} + \frac{T_1 - T_0}{2}$

Synchronisation interne : algorithme de Berkeley

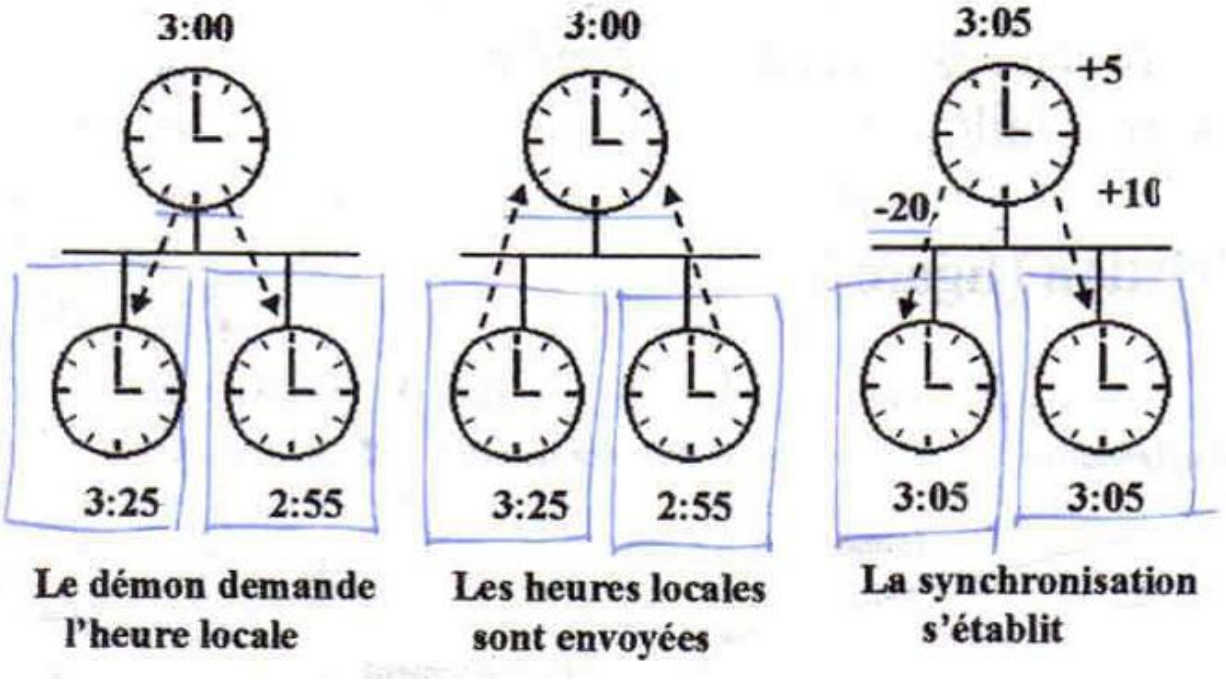
Problème : synchronisation interne d' un groupe de machines

- Un coordinateur demande aux participants (par diffusion) la valeur courante de leur horloge
- Le coordinateur estime l'heure locale de chaque participant au moyen du temps d'aller-retour (cf Cristian), et calcule la moyenne t_m de ces heures
- Le coordinateur demande aux participants de régler leur horloge sur cette heure moyenne (un participant en avance ne retarde pas son horloge, mais la ralentit pour atteindre progressivement la valeur fixée)

La précision du protocole dépend de l'estimation du temps d'aller-retour.
L'idéal est qu'il soit le même pour tous ; en pratique, on fixe un délai maximal et on ne tient pas compte des sites qui dépassent ce délai

Panne du coordinateur : on élit un autre coordinateur (cf plus loin : élection)
Le coordinateur élimine les valeurs trop déviantes

Algorithme de Berkeley

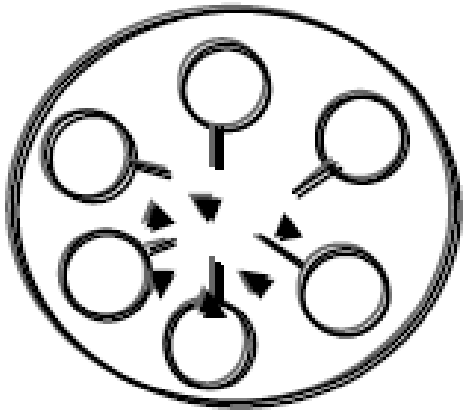


Le serveur de référence appelle les autres machines (inverse de Christian), il utilise un daemon.

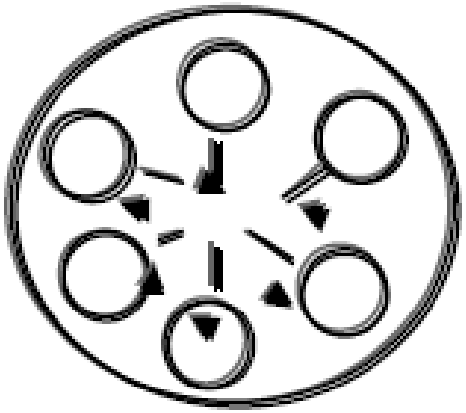
- Le daemon reçoit les réponses.
- Le daemon calcule un temps moyen, serveur compris, il ajuste le serveur si nécessaire et demande aux autres machines de s'ajuster en donnant les directives (+t1, t2, etc.).

Algorithmes basés sur la moyenne

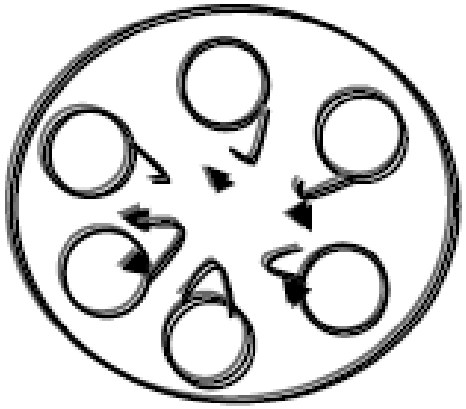
Christian et Berkeley sont des algorithmes centralisés. Il existe néanmoins des algorithmes répartis basés sur le fait que toutes les machines « broadcast » leurs temps régulièrement et chaque machine reçoit le temps de toutes les autres (ou une partie des autres), fait sa moyenne et calcule son temps.



Chaque machine diffuse son heure au début d ’une période de temps



Chaque machine démarre un temporisateur et collecte tous les messages arrivés durant un intervalle



Chaque machine exécute un même algorithme à partir des n données reçues pour l ’obtention d ’une heure moyenne

Exclusion-mutuelle

Rappel

- Une ressource partagée ou une section critique n'est accédée que par un processus à la fois
- Un processus est dans 3 états possibles, par rapport à l'accès à la ressource
  - ✓**Demandeur**
  - ✓**Dedans**
  - ✓**Dehors**
- Changement d'état par un processus
  - ✓De *dehors à demandeur*
  - ✓De *dedans à*
- Le passage de l'état *demandeur à l'état dedans est géré par le* système et/ou l'algorithme de gestion d'accès à la ressource (Couche middleware)

Rappel exclusion mutuelle

L'accès en exclusion mutuelle doit respecter deux propriétés

Sûreté (safety) : au plus un processus est à la fois dans la section critique (dans l'état *dedans)*

Vivacité (liveness) : tout processus demandant à entrer dans la section critique (à passer dans l'état *dedans) y entre en un* temps fini

Exclusion mutuelle distribuée

Plusieurs grandes familles de méthodes

Contrôle par un serveur
qui centralise les demandes d'accès à la ressource partagée

Contrôle par jeton

- un jeton unique circule sur l'ensemble des sites et donne le droit à son possesseur d'entrer en section critique. L'unicité du jeton assure la sûreté.

Contrôle par permission
le site demandeur doit recevoir l'accord d'un ensemble d'autres sites pour accéder à la section critique.

Contrôle par serveur

Principe général

- Un serveur centralise et gère l'accès à la ressource

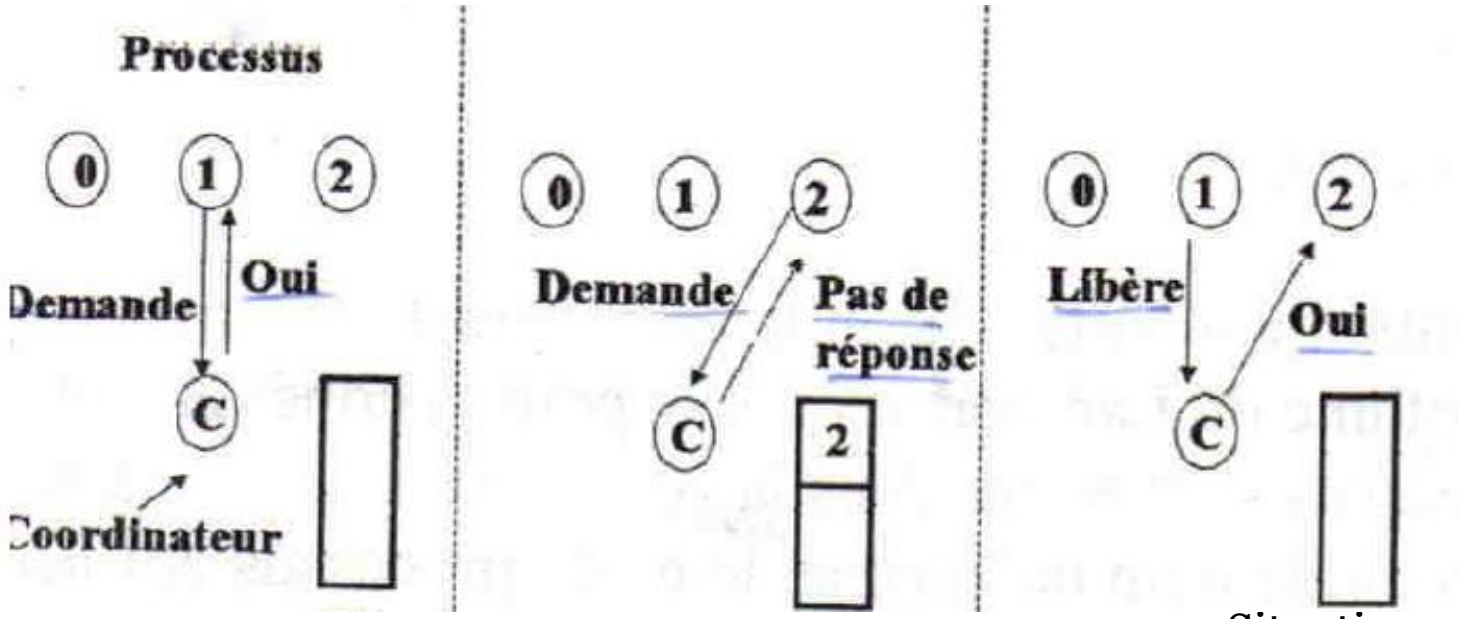
Algorithme

- Un processus voulant accéder à la ressource (quand il passe dans l'état *demandeur) envoie une requête au serveur*
- Quand le serveur lui envoie l'autorisation, il accède à la ressource (passe dans l'état *dedans)*

  ✓Il informe le serveur quand il relâche la ressource (passe dans l'état *dehors)*
- Le serveur reçoit les demandes d'accès et envoie les autorisations d'accès aux processus demandeurs

  ✓Avec par exemple une gestion FIFO : premier processus demandeur, premier autorisé à accéder à la ressource

Un algorithme centralisé implanté en réparti



Situation 1 Situation 2 Situation 3

- **Peu équitable (dépend de la vitesse de communication avec le coordinateur)**
- **Peu robuste**
- **Mais mise en oeuvre simple !**

Contrôle par permission

Méthodes par permission

- Un processus doit avoir l'autorisation des autres processus pour accéder à la ressource

Deux modes

- Permission individuelle : un processus peut donner sa permission à plusieurs autres à la fois
- Permission par arbitre : un processus ne donne sa permission qu'à un seul processus à la fois

  ✓Les sous-ensembles sont conçus alors tel qu'au moins un processus soit commun à 2 sous-ensembles : il joue le rôle d'arbitre

Permission individuelle

- Algorithme de [Ricart & Agrawala, 81]
- Chaque processus demande l'autorisation à tous les autres (sauf lui par principe)
- Se base sur une horloge logique (Lamport) pour garantir le bon fonctionnement de l'algorithme

Un algorithme distribué (Ricart & Agrawala-1981): Principe

· Quand un processus veut entrer dans la zone critique, il construit un message contenant le nom de la zone, son n° du processus et l'heure courante.
· Il envoie le message à tous les processus, lui inclus.
· Le message sera accusé (ACK) pour maintenir la fiabilité du mécanisme. La communication de groupe peut être utilisée.
· Les processus récepteurs répondront en fonction de leur état, lié à la zone en question, il y a 3 cas possibles :
1. Le récepteur n'est pas dans la zone et ne veut pas y entrer, il répond OK au demandeur.
2. Le récepteur est déjà dans la zone, il ne répond pas. Il charge la demande dans sa file d'attente.
3. Le récepteur veut aussi entrer dans la zone. Il a déjà envoyé les messages.

Un algorithme distribué (Ricart & Agrawala-1981): Principe

Il compare l'heure du message qu'il vient de recevoir avec l'heure stockée dans le message qu'il a déjà envoyé. Si l'heure du message reçu est inférieure à la sienne, il répond OK, sinon (si son heure est inférieure) il place la demande dans sa file et ne répond pas.

· Quand sa demande est partie, le processus attend son acceptation par tous les autres processus, ensuite il entre dans la zone critique. Quand il en sort, il répond OK aux autres processus qui sont dans sa file d'attente et les enlève de sa file d'attente.

Un algorithme distribué (Ricart & Agrawala-1981)

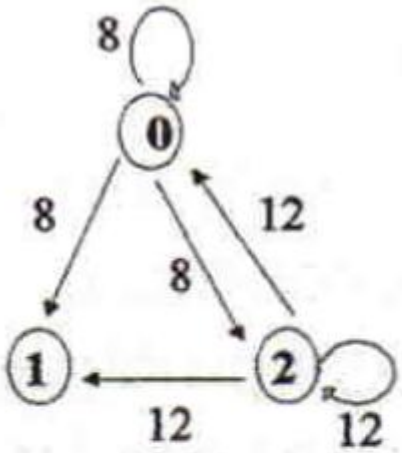
•Cet algorithme nécessite un ordre total de tous les évènements dans le système.

•Dans une série d'évènements il est essentiel que l'ordre dans lequel ils ont été demandés soit non ambigu.

•L'algorithme de Lamport, vu précédemment, sera utilisé pour préserver cet ordre et fournir les *timestamp nécessaires à l'exclusion mutuelle distribuée*.

Un algorithme distribué (Ricart & Agrawala-1981)

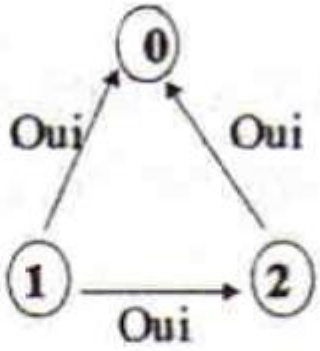
Phase 1



Les processus 0 et 2 veulent entrer dans la même zone critique. Les messages sont envoyés.

a)

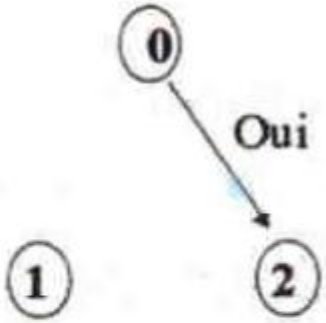
Phase 2



Le processus 0 reçoit les approbations il entre dans la zone critique

b)

Phase 3



Le processus 0 a terminé, il envoie l'approbation au processus 2 qui entre dans la zone critique

c)

Inconvénients de cet algorithme :

•Couteux en nombre de messages par SC: 2(N-1)
•Robuste (mais, savoir exactement combien de processus sont vivants (N doit constamment être exacte)

•Il n'y a plus un seul point de panne, mais n. En particulier à cause des non réponses qui peuvent être interprétées comme un refus alors qu'un processeur peut-être en panne et non apte à répondre. Les autres processus vont attendre pour rien.

•Amélioration possible : avoir une majorité (à déterminer) de OK pour rentrer en zone critique. Bien sûr un processus ne pourra pas donner la même permission à un deuxième processus tant que le premier n'a pas libérer la ressource.

Algorithme de **Ricart & Agrawala-1981**

.Respect des propriétés

- Sûreté : vérifiée
- Vivacité : assurée grâce aux datations et aux priorités associées

Permission individuelle

- amélioration de l'algorithme de [Ricart & Agrawala, 81]

- **[Carvalho & Roucairol, 83]**
  ✓Si *Pi veut accéder plusieurs fois de rang à la ressource partagée* et si *Pj entre 2 accès (ou demandes d'accès) de Pi n'a pas* demandé à accéder à la ressource
    ❖Pas la peine de demander l'autorisation à *Pj car on sait alors qu'il donnera* par principe son autorisation à *Pi*
    ❖Limite alors le nombre de messages échangés

Carvalho & Roucairol, 83

Principe de l'algorithme

Soit le cas où P_i demande l'accès à la Section Critique plusieurs fois de suite (état ***demandeur*** plusieurs fois de suite) alors que P_j n'est pas intéressé par celle-ci (état ***dehors***)

- Avec l'algorithme de Ricart et Agrawala, P_i demande la permission de P_j à chaque nouvelle demande d'accès à la Section Critique
- Avec l'algorithme de Carvalho et Roucairol, puisque P_j a donné sa permission à P_i, ce dernier la considère comme acquise jusqu'à ce que P_j demande sa permission à P_i : P_i ne demande qu'une fois la permission à P_j

Permission individuelle

amélioration de l'algorithme de [Ricart & Agrawala, 81]

- **[Chandy & Misra, 84],** améliorations tel que
  - ✓Les processus ne voulant pas accéder à la ressource et qui ont déjà donné leur permission ne reçoivent pas de demande de permission
  - ✓Horloges avec des datations bornées (modulo *m)*
  - ✓Pas d'identification des processus

Permission par arbitre

- Un processus ne donne qu'une permission à la fois

Il redonnera sa permission à un autre processus quand le processus a qui il avait donné précédemment la permission lui a indiqué qu'il a fini d'accéder à la ressource

- La sûreté est assurée car

✓ Les sous-ensemble de processus à qui un processus demande la permission sont construits tel qu'ils y ait toujours au moins un processus commun à 2 sous-ensemble

✓ Un processus commun à 2 sous-ensembles est alors arbitre

❖ Comme il ne peut donner sa permission qu'à un seul processus, les processus de 2 sous-ensembles ne peuvent pas tous donner simultanément la permission à 2 processus différents

❖ C'est donc ce processus commun qui détermine à qui donner la ressource

Permission par arbitre

- **Algorithme de [Maekawa, 85]**
  - Chaque processus *Pi* possède un sous-ensemble *Ri* d'identificateurs de processus à qui *Pi* demandera l'autorisation d'accéder à la ressource
    - $\forall i,j \in [\ 1..N\ ] : Ri \cap Rj \neq \varnothing$
      - Deux sous-ensembles de 2 processus différents ont obligatoirement au moins un élément en commun (le ou les arbitres)
      - Cela rend donc inutile le besoin de demander la permission à tous les processus, d'où les sous-ensembles Ri ne contenant pas tous les processus
    - $\forall i : |\ Ri\ | = K$
      - Pour une raison d'équité, les sous-ensembles ont la même taille pour tous les processus
    - $\forall i$: i est contenu dans D sous-ensembles
      - Chaque processus joue autant de fois le rôle d'arbitre qu'un autre processus

Permission par arbitre

Algorithme de [Maekawa, 85] (suite)

Solution optimale en nombre de permissions à demander et de messages échangés

Fonctionnement de l'algorithme

Chaque processus possède localement

- Une variable *vote permettant de savoir si le processus a déjà* voté (a déjà donné sa permission à un processus)
- Une file *file d'identificateurs de processus qui ont demandé la* permission mais à qui on ne peut la donner de suit
- Un compteur *réponses du nombre de permissions reçues*

Initialisation

- État non demandeur, vote = faux et file = ∅, réponses = 0

Permission par arbitre

- **Algorithme de [Maekawa, 85] (suite)**
  - Quand processus *Pi* veut accéder à la ressource
    - réponses = 0
    - Envoie une demande de permission à tous les processus de *Ri*
    - Quand *réponses* = | *Ri* |, *Pi* a reçu une permission de tous, il accède alors à la ressource
    - Après l'accès à la ressource, envoie un message à tous les processus de *Ri* pour les informer que la ressource est libre
  - Quand processus *Pi* reçoit une demande de permission de la part du processus *Pj*
    - Si *Pi* a déjà voté (vote = vrai) ou accède actuellement à la ressource : place l'identificateur de *Pj* en queue de *file*
    - Sinon : envoie sa permission à *Pj* et mémorise qu'il a voté
      - vote = vrai

Permission par arbitre

- **Algorithme de [Maekawa, 85] (suite)**

  - Quand *Pi* reçoit de la part du processus *Pj* un message lui indiquant que *Pj* a libéré la ressource
    - Si *file* est vide, alors vote = faux
      - *Pi* a déjà autorisé tous les processus en attente d'une permission de sa part
    - Si *file* est non vide
      - Retire le premier identificateur (disons *k*) de la file et envoie à *Pk* une permission d'accès à la ressource
      - *vote* reste à vrai

Permission par arbitre

- **Algorithme de [Maekawa, 85] probleme!**

- La vivacité n'est pas assurée par cet algorithme car des cas d'interblocage sont possibles
- Pour éviter ces interblocages, améliorations de l'algorithme en définissant des priorités entre les processus

✓ En datant les demandes d'accès avec une horloge logique

✓ En définissant un graphe de priorités des processus

Méthode par jeton

Principe général

- Un jeton unique circule entre tous les processus
- Le processus qui a le jeton est le seul qui peut accéder à la section critique

Respect des propriétés

- Sûreté : grâce au jeton unique
- Vivacité : l'algorithme doit assurer que le jeton circule bien entre tous les processus voulant accéder à la ressource

Plusieurs versions

- Anneau sur lequel circule le jeton en permanence
- Jeton affecté à la demande des processus

Méthode par jeton

Algorithme de [Le Lann, 77]

- Un jeton unique circule en permanence entre les processus via une topologie en anneau
- Quand un processus reçoit le jeton

✓ S'il est dans l'état *demandeur : il passe dans l'état dedans et* accède à la ressource

✓ S'il est dans l'état *dehors, il passe le jeton à son voisin*

- Quand le processus quitte l'état *dedans, il passe le jeton à* son voisin

Respect des propriétés

- **Sûreté** : via le jeton unique qui autorise l'accès à la ressource
- **Vivacité** : si un processus lâche le jeton (la ressource) en un temps fini et que tous les processus appartiennent à l'anneau

Méthode par jeton

Algorithme de [Le Lann, 77]

Inconvénients

- Nécessite des échanges de messages (pour faire circuler le jeton) même si aucun site ne veut accéder à la ressource
- Temps d'accès à la ressource peut être potentiellement relativement long
- Si le processus *i+1 a le jeton et que le processus i veut accéder à la* ressource et est le seul à vouloir y accéder, il faut quand même attendre que le jeton fasse tout le tour de l'anneau
- Perte du jeton
- Problème de la panne d'un pro cessu

Avantages

- Très simple à mettre en oeuvre
- Intéressant si nombreux processus demandeurs de la ressource
- Jeton arrivera rapidement à un processus demandeur
- Équitable en terme de nombre d'accès et de temps d'attente
- Aucun processus n'est privilégié

Méthode par jeton

Variante de la méthode du jeton

- Au lieu d'attendre le jeton, un processus diffuse à tous le fait qu'il veut obtenir le jeton
- Le processus qui a le jeton sait alors à qui il peut l'envoyer
- Évite les attentes et les circulations inutiles du jeton

Algorithme de [Ricart & Agrawala, 83]

- Soit *N processus avec un canal bi-directionnel entre* chaque processus
  - ✓ Canaux fiables mais pas forcément FIFO
- Localement, un processus *Pi possède un tableau nbreq*, de taille *N*
- Pour *Pi, nbreq [j] est le nombre de requêtes d'accès que le processus Pj a fait et que Pi connaît (par principe il les connaît toutes)*

Méthode par jeton

- Algorithme de [Ricart & Agrawala, 83] (suite)
  - Le jeton est un tableau de taille *N*
    - *jeton [i]* est le nombre de fois où le processus *Pi* a accédé à la ressource
    - La case *i* de *jeton* n'est modifiée que par *Pi* quand celui-ci accède à la ressource
  - Initialisation
    - Pour tous les sites *Pi* : $\forall j \in [1 .. N] : nbreq[j] = 0$
    - Jeton : $\forall j \in [1 .. N] : jeton[j] = 0$
    - Un site donné possède le jeton au départ
  - Quand un site veut accéder à la ressource et n'a pas le jeton
    - Envoie un message de requête à tous les processus

Méthode par jeton

- Algorithme de [Ricart & Agrawala, 83] (suite)
  - Quand processus *Pj* reçoit un message de requête venant d
    - *Pj* modifie son *nbreq* localement : nbreq [i] = nbreq [i] + 1
      - *Pj* mémorise que *Pi* a demandé à avoir la ressource
    - Si *Pj* possède le jeton et est dans l'état *dehors*
      - *Pj* envoie le jeton à *Pi*
  - Quand processus récupère le jeton
    - Il accède à la ressource (passe dans l'état *dedans*)
  - Quand Pi libère la ressource (passe dans l'état *dehors*)
    - Met à jour le jeton : jeton [i] = jeton [i] + 1
    - Parcourt *nbreq* pour trouver un *j* tel que : nbreq [j] > jeton [j]
      - Une demande d'accès à la ressource de *Pj* n'a pas encore été satisfaite : *Pi* envoie le jeton à *Pj*
      - Si aucun processus n'attend le jeton : *Pi* le garde

Méthode par jeton

- Algorithme de [Ricart & Agrawala, 83], respect des propriétés
  - Sûreté : seul le processus ayant le jeton accède à la ressource
  - Vivacité : assurée si les processus distribuent équitablement le jeton aux autres processus
    - Méthode de choix du processus qui va récupérer le jeton lorsque l'on sort de l'état dedans
      - *Pi* parcourt *nbreq* à partir de l'indice i+1 jusqu'à *N* puis continue de 1 à i-1
      - Chaque processus teste les demandes d'accès des autres processus en commençant à un processus spécifique et différent de la liste
      - Évite que par exemple tous les processus avec un petit identificateur soient servis systématiquement en premier

Algorithmes d'élection

Election

Principe :

Choisir un et un seul leader parmi un ensemble de processus et le faire connaître de tous.

Hypothèses :

L'élection peut être déclenchée par un processus arbitraire ou éventuellement par plusieurs processus.

Leader : le processus qui a le plus grand numéro.

Propriétés :

Sûreté (safety) : un seul processus doit être élu
Vicacité(liveness) :un processus doit etre élu en un temps fini

Définition et utilisations

- *En pratique :*
  - le nombre des processus est connu a priori
  - les processus sont identifiés par un nombre (n° de processus, de site, etc)

Usage

- *Choix d'un processus maître (notamment après défaillance)*
  - pour la coordination d'un ensemble de processus
  - la régénération d'une information perdue (ex : jeton)
  - le contrôle de concurrence (séquencement)

Algorithme Brute de Force

Hypothèses :

Il y a aucune perte de messages
Il y a une borne connue sur le temps de communications.

Principe :

Les demandes d'élection sont diffusées par inondation.
Un processus répond a ceux de numéro inferieur au sien.
Un processus qui ne reçoit aucune réponse constate qu'il est élu.

Complexité :

$O(n^2)$ messages au pire des cas.

Algorithme Brute de Force

Déclanchement :

Quand un processus P s'aperçoit que le coordinateur ne répond plus à ses requêtes (time-out sur *TEMPO*), il lance l'algorithme d'élection

Lancement d'une élection par P :

Envoi d'un message `ELECTION` à tous les autres processus dont le numéro est plus grand que le sien

Réception d'un message `ELECTION` depuis P par un processus Q :

- Le processus Q envoie un message `ACK` à P lui signifiant qu'il est actif
- A son tour Q, lance une élection si ce n'est pas déjà fait

Algorithme Brute de Force

Sur le processus P :

- Si aucun processus ne lui répond avant *TEMPO*, P gagne l'élection et devient le coordinateur
- Si un processus de numéro plus élevé répond, c'est lui qui prend le pouvoir. Le rôle de P est terminé.

Annonce de l'élu

Le nouveau coordinateur envoie un message à tous les participants pour les informer de son rôle. L'application peut alors continuer à s'exécuter

Algorithme Brute de Force

Réveil d'un processus inactif

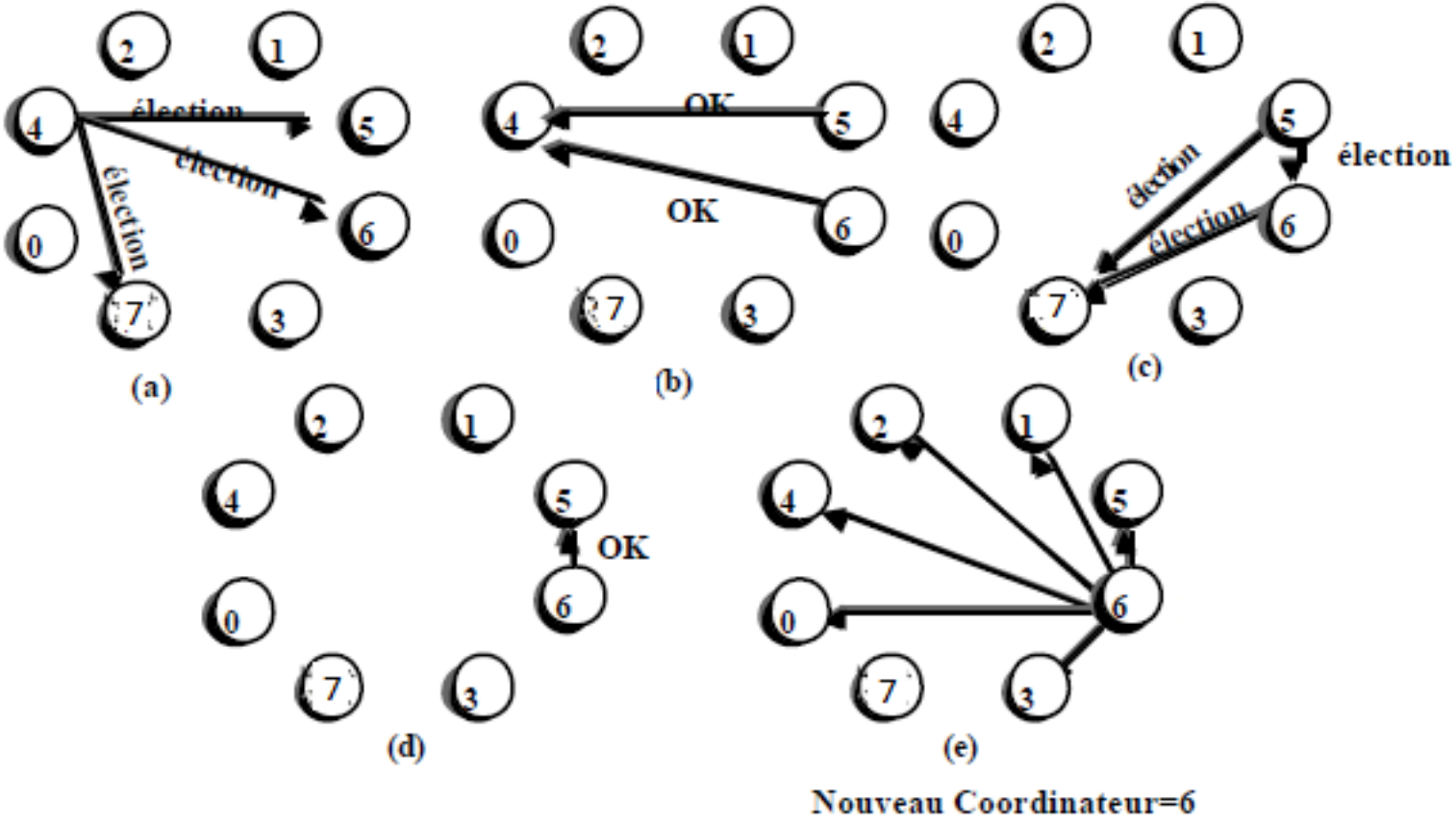
- Déclenche une élection
- S'il détient le plus grand numéro de processus en cours de fonctionnement, il gagne l'élection et devient le nouveau coordinateur

Algorithme Brute de Force

Déroulement de l'algorithme

- Application composée de 8 processus numérotés de 0 à 7
- Processus de numéro 7 tombe en panne, Processus de numéro 4 est le premier à détecter cette panne

Algorithme Brute de Force**(Garcia-Molina 1982)**



Algorithme sur un anneau

Principe :

- plusieurs sites peuvent démarrer le processus d‘élection
- Chaque site qui commence une élection envoie un jeton en précisant sa valeur
- Lors qu'un site reçoit un jeton de valeur inferieure, il le supprime

Algorithme utilisant un anneau [Chang-Roberts 79]

Principe

- *Extinction progressive des messages par filtrage*

Réalisation

- *Anneau virtuel unidirectionnel*
- *Programme du site i :*
- (chaque site i dispose d'un pointeur vers son successeur succ[i])
- plusieurs candidats simultanés possibles

Idée :

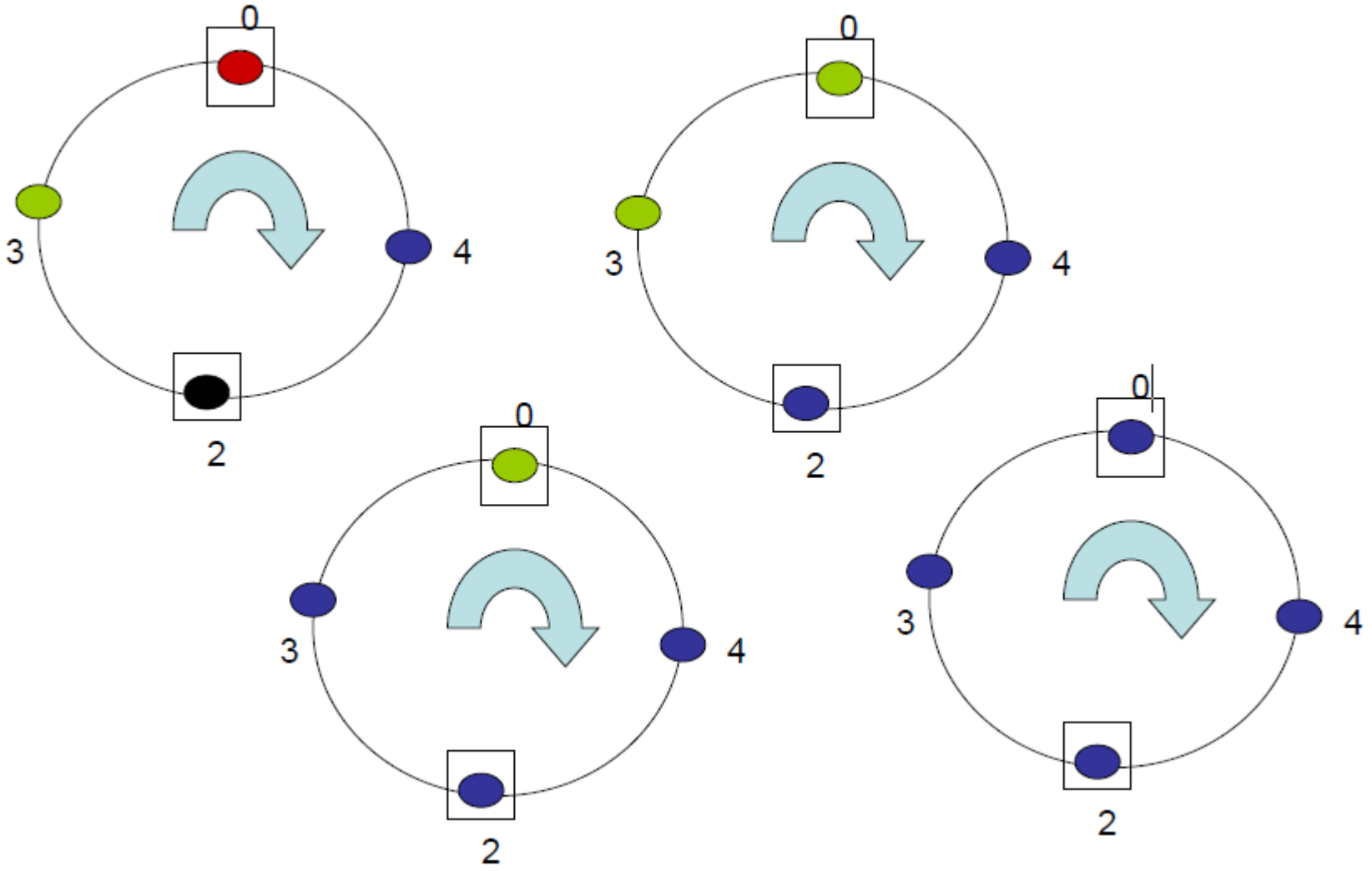
chaque candidat diffuse autour de l'anneau sa candidature; le processus ayant l'identifiant max gagne

Algorithme utilisant un anneau [Chang-Roberts 79]

Hyp othè et principe

- Supprime les jetons des processus qui ne sont pas élus : ceux qui ont un numéro de processus plus petit
- Un initiateur perd quand il reçoit un jeton qui porte un numéro supérieur
- Un processus devient leader lorsqu'il reçoit son jeton

Algorithme utilisant un anneau [Chang-Roberts 79]



Algorithme utilisant un anneau [Chang-Roberts 79]

Candidature (site pi):

candidat$_i$ = true

send (succ[i], ELECT, i)

R1: receive(pi, ELECT, j)

switch

j > i : send (succ[i], ELECT, j) *// j prioritaire*

j < i : if not candidat$_i$ *// j non prioritaire*

candidat$_i$ = true

send (succ[i], ELECT, i)

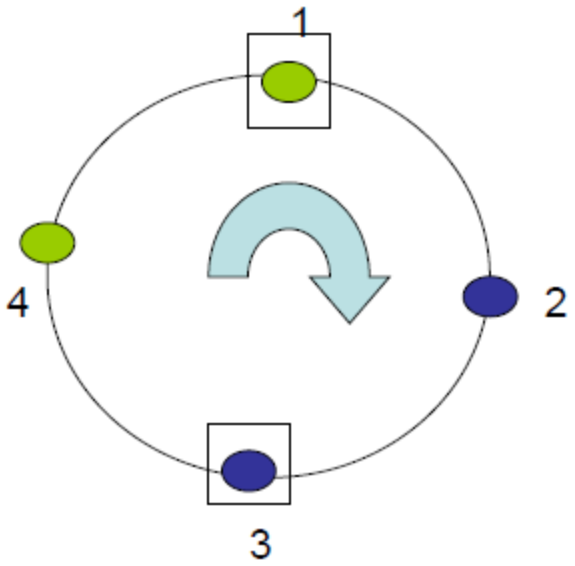
j = i : send (all, ELECTED, i) *// site i élu*

O(N log N) messages en moyenne, O(N^2) au pire

Compléxité *Chang-Roberts 79]*

Le meilleur cas : O(n)

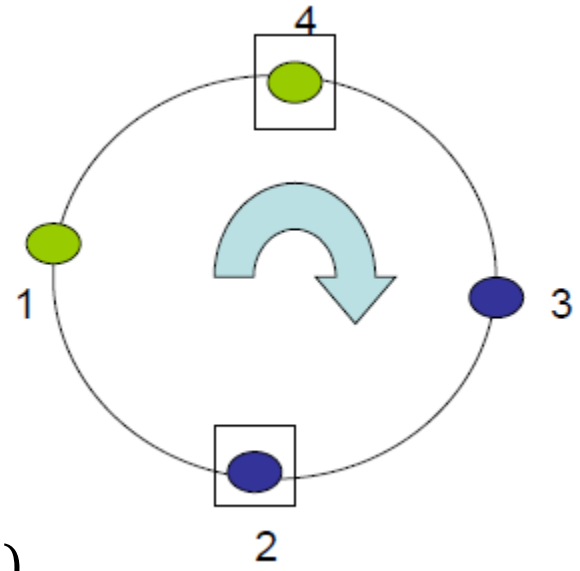
Les identifiants sont ordonnes dans l'ordre croissant autour de l'anneau



Compléxité *Chang-Roberts 79]*

Le pire cas : $O(n^2)$

· Les identifiants sont ordonnes dans l'ordre décroissant autour de l'anneau
· L'identifiant « i » visite i nœuds avant de décider son status



En moyenne : O(n log n)

Transactions atomiques

Introduction

Transaction: signifie la suite des instructions dont on veut exécuter l 'intégralité
=> transaction atomique
=> délimiter par des pseudo-instructions

Transaction répartie: si par exemple C1 et C2 sont sur des sites différents (et donc les actions sur ces objets aussi). La transaction doit se réaliser tout en étant atomique
=> coopération des sites

Transaction concurrente: si par exemple une autre transaction manipule C1 et/ou C2, en particulier en modification.
=> s 'assurer que leur résultat est comparable avec ce qu'il se serait passé si les 2 transactions s'exécutaient en séquence.

Mais par souci de performance, trouver des techniques pour qu'elles s 'exécutent le plus possible en parallèle : **CONTROLE de CONCURRENCE**

Modèle de transaction

TRANSACTION

On appelle transaction l'unité de traitement séquentiel, exécutée pour le compte d'un processus, appliquée à une mémoire stable cohérente, restituant une mémoire stable cohérente

PROPRIÉTÉS

-**Atomicité** : Une transaction est une unité logique indivisible de traitement qui est soit complètement exécutée soit complètement abandonnée

-**Sérialisabilité** : Les transactions concurrentes n ’interfèrent pas entre elles (en particulier on ne veut pas voir les états intermédiaires non encore validés par d’autres)

- **Permanence** : Si une transaction est validée, tous les changements par elle sur la mémoire stable sont devenus définitifs

Propriétés ACID: Atomicité / Cohérence / Isolation / Durabilité

Modèle de transaction

Propriétés ACID

Atomicité
tout ou rien

Consistance
cohérence sémantique

Isolation
pas de propagation de résultats non validés

Durabilité
persistance des effets validés

Modèle de transaction

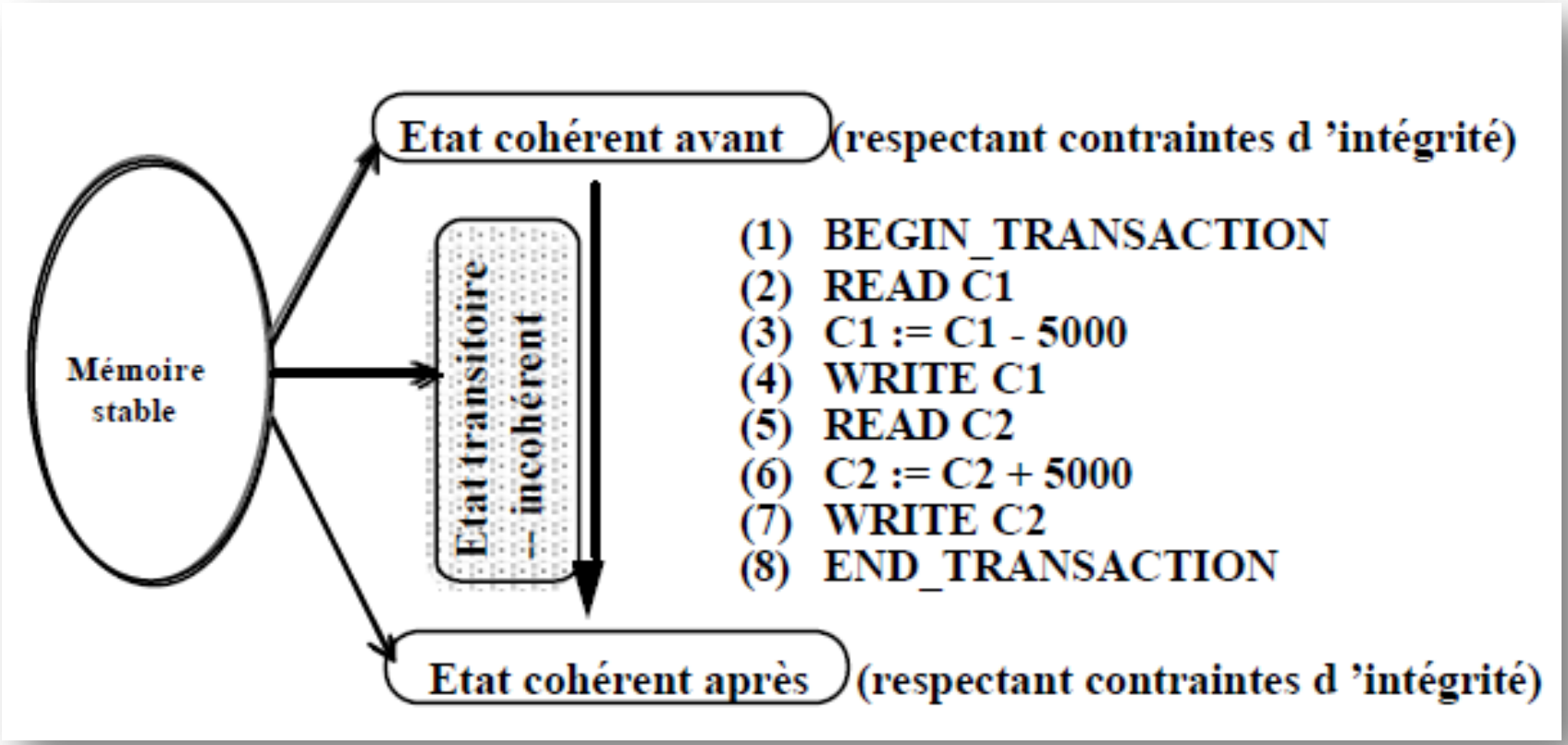


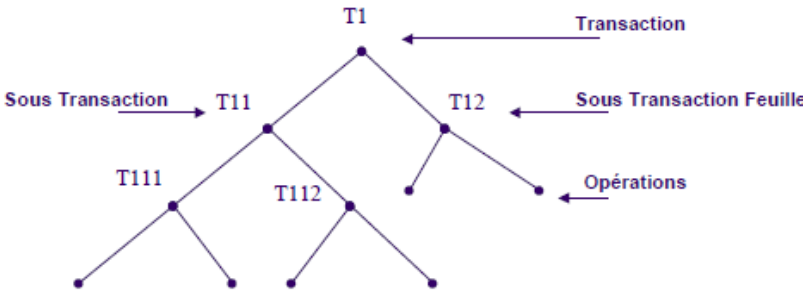
Illustration de la notion de « mise-à-jour sûre » par une transaction

Modèle de transaction: Transactions imbriquées

Une transaction peut contenir des sous-transactions appelées transactions imbriquées (peu courant; plutôt modèle plat -- flat)

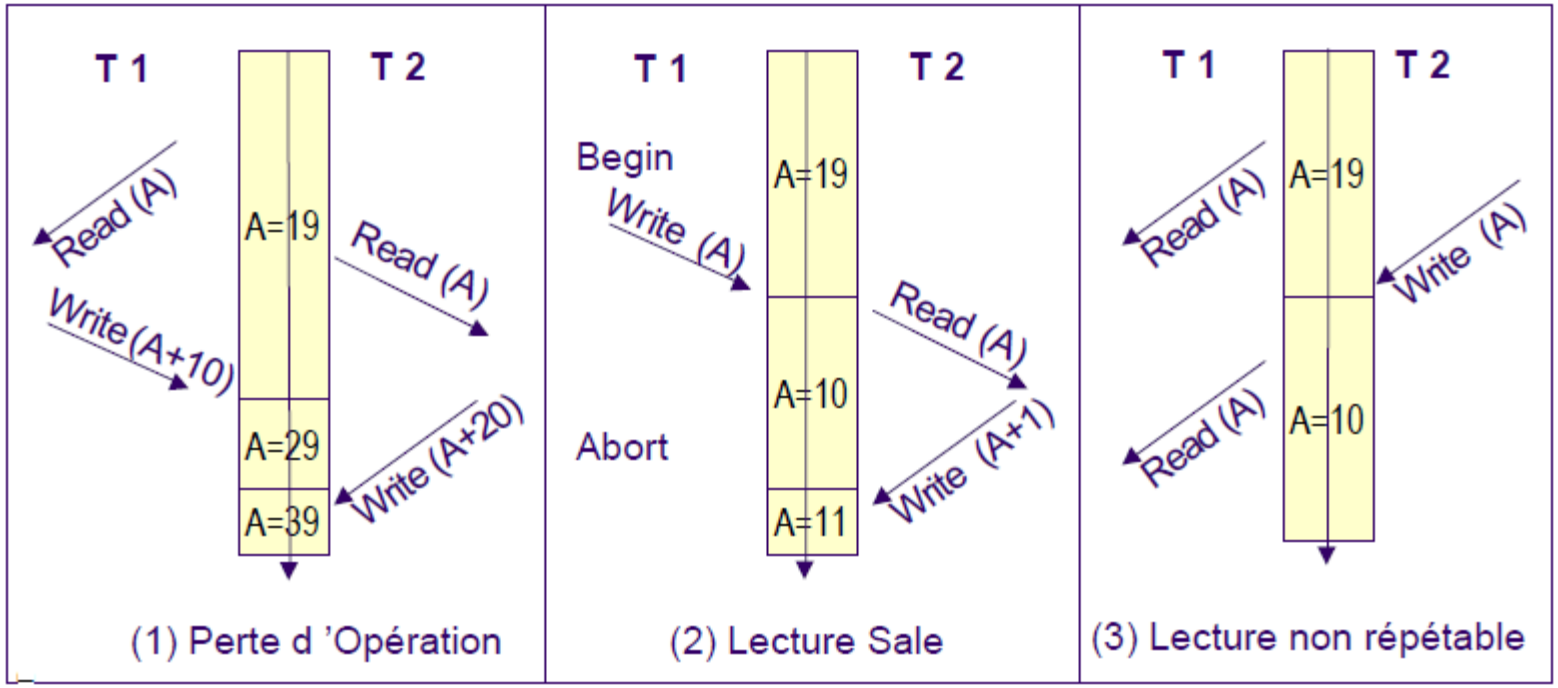
Règles de fonctionnement

- Une sous transaction démarre après la transaction mère et se termine avant elle
- L 'abandon d'une sous transaction entraîne :
  - l 'abandon de ses sous transactions descendantes,
  - pas nécessairement l 'abandon de ses ancêtres
- La validation d'une sous transaction est conditionnée à la validation de sa transaction mère
- Une sous transaction est atomique et isolée de toutes les sous transactions qui n 'appartiennent pas à sa descendance



Notion de cohérence

- Une exécution concurrente non contrôlée de plusieurs transactions crée des incohérences



Techniques de contrôle de concurrence

- **Exécution sérielle**

T1 puis T2 puis T3 puis T4
T2 puis T1 puis T3 puis T4
…

- **Sérialisabilité**

entrelacement des actions des transactions tel que le résultat est équivalent à celui d ’une des exécutions Sérielles

- **Méthodes**

Pessimiste : conflits probables
Optimiste : conflits peu probables

a) Caractérisation d ’Ordonnancements sérialisables

utilisation d’un graphe de dépendance entre transactions concurrentes

Relation de dépendance

Soit S un ordonnancement sans opération simultanée. On dit que Ti précède Tj (noté Ti<Tj) dans S si et seulement s ’il existe deux opérations non permutables Oi et Oj (parce que conflictuelles) telles que Oi est exécutée par Ti avant Oj par Tj

Théorème (Papadimitriou79)

Une condition suffisante pour qu’un ordonnancement soit sérialisable est que le graphe de précédence associé soit sans circuit

b) Techniques de Verrouillage (contrôle Pessimiste)

Opérations protocolaires

- **LOCK(g,M)** permet à une transaction de demander de poser un verrou de type M sur l ’objet g
- **UNLOCK(g)** permet à une transaction de demander de lever le verrou qu’elle a posé sur l ’objet g

Gestion de conflits:

Le système maintient une table d ’allocation de verrous sur les objets. Cette information est utilisée pour détecter les conflits.
La transaction à l ’origine du conflit attend la libération de l ’objet par la (les) transaction(s) concurrente(s).
Si le gestionnaire de verrous est distant on lui envoie un message transportant le LOCK ou le UNLOCK.
Le graphe des dépendances représente donc aussi un graphe des attentes.
Un circuit dans ce graphe signifiera donc l ’existence d ’un interblocage des transactions du circuit.

b) Techniques de Verrouillage: Condition de sérialisation

Les règles de construction suivantes représentent une condition suffisante pour garantir la propriété de sérialisabilité (Eswaran 76)

•**la transaction est bien formée**: tout accès à un objet est précédé d ’une opération de verrouillage compatible avec le mode d ’accès

•**le verouillage est à 2 phases**: aucune demande de verrouillage n ’est acceptée après la première demande de libération

c) Autres techniques de Contrôle de Concurrence

Contrôle de Concurrence optimiste (Kung et Robinson, 1981):

Idée de base

1) Allez de l ’avant et faites tout ce que vous voulez, sans faire attention à ce que les autres font

2) S ’il arrive un problème, on s ’en occupera plus tard

c) Autres techniques de Contrôle de Concurrence

Principe de fonctionnement

- considérer que chaque transaction se compose de deux étapes :
    - une étape de lecture et calcul dans un espace privé
    - une étape d'écriture réelle dans la base (commitment) (qui pourra donc échouer si concurrence)
- ordonner les transactions selon le moment où elles terminent la première étape de lecture et de calcul
- durant l ’ écriture réelle, contrôler que les accès conflictuels aux objets s'effectuent bien dans l'ordre ainsi obtenu

Evaluation

•Fonctionne bien si peu de conflits car conflits peuvent engendrer des abandons, et donc, des reprises ultérieures de transactions (beaucoup d ’abandons => grosse surcharge)

•Au contraire, les techniques pessimistes fondées par exemple sur le verouillage, et donc des attentes, est préférable pour des transactions longues.

c) Autres techniques de Contrôle de Concurrence

Technique pessimiste avec utilisation d'Estampillage (Reed 1983)

Estampille de transaction
valeur numérique unique associée à une transaction au moment de son commencement (BEGIN_TR.)
Mieux: au moment de l 'apparition d 'un premier conflit.
En réparti utiliser Horloges de Lamport

Estampille d 'objet par type d 'opération
valeur numérique associée à un objet mémorisant l 'estampille de la dernière transaction ayant opéré sur cet objet via cette opération

Principe

1- toute transaction a une estampille unique. Les estampilles engendrent un ordre total des transactions

2- une transaction ayant accédé à un objet doit stocker son estampille sur cet objet

3- une transaction voulant accéder à un objet doit comparer son estampille avec celle de l ’objet: si elle est plus jeune que celle ayant accédé en dernier à cet objet, elle obtient l ’accès sur l ’objet (et subit une attente si cet objet est en cours d ’utilisation)

4- dans le cas contraire, elle devra se suicider (l ’ordre d ’exécution indique que la transaction courante est « en retard »). La reprise se fera avec une nouvelle estampille.
=> Cette technique n ’engendre pas d ’interblocage car il y a immédiatement abandon de la transaction en retard.

d) Gestion des interblocages

Quatre stratégies de gestion

1. La politique de l ’autruche (ignorer le problème)

2. La détection (permettre aux interblocages de se produire, les détecter, puis tenter de les éliminer. Cas transactionnel: abandonner une transaction est possible alors que c ’est plus difficile pour un processus non transactionnel)

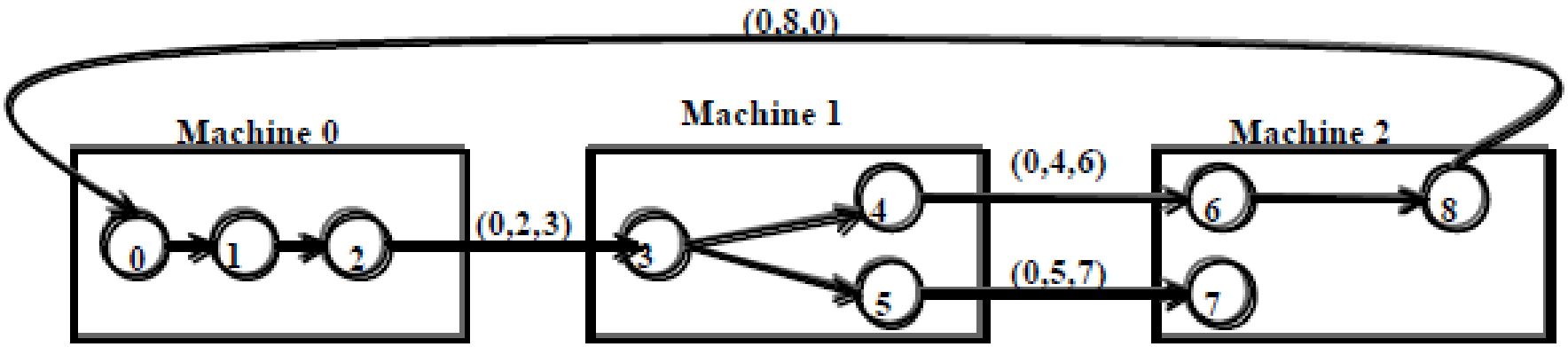
3. La prévention (rendre de façon statique les interblocages structurels impossibles)

4. L ’évitement (éviter les interblocages en distribuant les ressources avec précaution)

Détection des interblocages : algorithmes distribués

Ex: P0 se bloque sur P1 (P0 demande une ressource que P1 a verrouillé)
⇒une sonde (0, 0, 1) est émise , elle sera transmise de site en site.

Champ1: num du processus qui se bloque;
Champ2: émetteur de la sonde;
Champ3: récepteur de la sonde



Détection des interblocages : algorithmes distribués

Algorithme de Chandy-Misra-Haas (1983):

Principe d ’une sonde qui va parcourir le graphe des attentes. Si la sonde revient, alors, la nouvelle attente vient d ’engendrer un interblocage.
Guérison ? Abandonner la transaction « fautive », celle ayant le moins de verrous, la plus jeune ?

Détection/Guérison plus simple par Temporisation:

- fixer durée max d ’exécution pour une transaction
- si au bout du délai une transaction n ’a pas fini, suspicion qu’elle est interbloquée et donc abondon de la transaction

Prévention des interblocages : généralité

Principe

La prévention des interblocages consiste à construire prudemment le système de façon de rendre les interblocages structurellement impossibles. Autrement dit, elle consiste à supprimer l 'une des conditions qui rend possible l'interblocage

Contraintes d 'allocation

- un processus ne détient qu'une ressource à la fois
- un processus doit déclarer toutes les ressources dont il a besoin
- un processus doit relâcher les ressources qu'il détient pour pouvoir en obtenir une nouvelle

=> mal adapté pour verrouillage en 2 phases !

Ordonnancement des ressources

- préordonner toutes les ressources du système

- un processus doit acquérir les ressources dont il a besoin dans l 'ordre strictement croissant

Ordonnancement des transactions

- préordonner toutes les transactions par une estampille

- déterminer une politique d 'allocation des ressources adéquate permettant d 'éviter les interblocages (ex, Reed83)

Algorithmes de prévention de Rosenkrantz, Stearns et Lewis:

• L'idée de l'algorithme de Rosenkrantz, Stearns et Lewis est de prévenir l'interblocage en basant uniquement sur l'estampillage.

• Principe : lorsqu'une ressource r, utilisée par une transaction T1, est demandée par une transaction T2, les conflits sont résolus par comparaison de leurs estampilles.

• Il n'y a ici aucune annonce préalable : l'algorithme proposé peut donc s'appliquer lorsque les *accès aux ressources sont définis dynamiquement*.

• Il s'agit toujours d'un algorithme de prévention : la comparaison des estampilles a pour effet d'empêcher la formation de circuits dans le graphe des conflits.

• Deux techniques sont proposées : avec ou sans réquisition.

Algorithmes de prévention de Rosenkrantz, Stearns et Lewis:

Technique sans réquisition : « Wait-Die System»

• On considère deux transactions t1 et t2 et une ressource r utilisée par t1 et demandée par t2.

• A la création de chaque transaction une estampille lui est associée; soient e(t1) et e(t2) les estampilles des deux transactions

• L'algorithme auquel obéit l'allocateur situé sur le site de la ressource r est le suivant :

si e(t2) < e(t1) alors bloquer t2 /* wait */
sinon annuler t2 /* die */
fsi

⇒ si la transaction t2 est la plus ancienne elle reste bloquée jusqu'à ce que t1 libère la ressource.

⇒ sinon t2 est annulée et sera redémarrée ultérieurement avec le même numéro d'estampille

⇒ cet algorithme évite la famine des transactions annulées

Algorithmes de prévention de Rosenkrantz, Stearns et Lewis:

Technique avec réquisition : « Wound-Wait System»

• Une solution consiste à satisfaire systématiquement les demandes de transactions plus anciennes et donc d'annuler les transactions plus jeunes qui utilisent les ressources : il y a donc réquisition.

• L'algorithme de l'allocateur situé sur le site de la ressource r est le suivant (r est utilisée par t1 et demandée par t2) :

si e(t2) < e(t1) alors annuler t1 /* Wound */
Sinon bloquer t2 /* Wait */
Fsi

• Une transaction n'attend jamais une ressource utilisée par une transaction plus jeune.

Algorithmes de prévention de Rosenkrantz, Stearns et Lewis:

Conclusion :

• Ces algorithmes ont l'avantage d'être simples

• Peuvent entraîner l'annulation de transactions qui n'auraient pas conduit à un interblocage

• Intéressant si peu de conflits sur une même ressource

Contrôle de l'atomicité

• **But**:
pouvoir « défaire » ou « refaire » ce qu 'une transaction a fait, en cas de défaillance cad, garantir la propriété d 'atomicité de la transaction.

• Défaillance parce que:

  • abandon engendré par le controle de concurrence
  • abandon programmé explicitement, ou erreur de programmation
  • panne du site ou des stockages de données

• **Principe**:
  • Garder trace de ce qui est exécuté par les transactions via la redondance : espace privé (copie de l 'espace global) sur lequel travaille la transaction, ou bien, utilisation d 'un journal de toutes les actions qui ont lieu sur l 'espace global (journal Avant ou journal Après)
  • Procéder à l 'exécution d 'une transaction en 2 phases: calcul puis validation initiée en fin de transaction. Si transaction est répartie, il faut coordonner la validation de tous les sites impliqués : validation en plusieurs phases

Contrôle de l’atomicité : « défaire » ou « refaire » en cas de défaillances

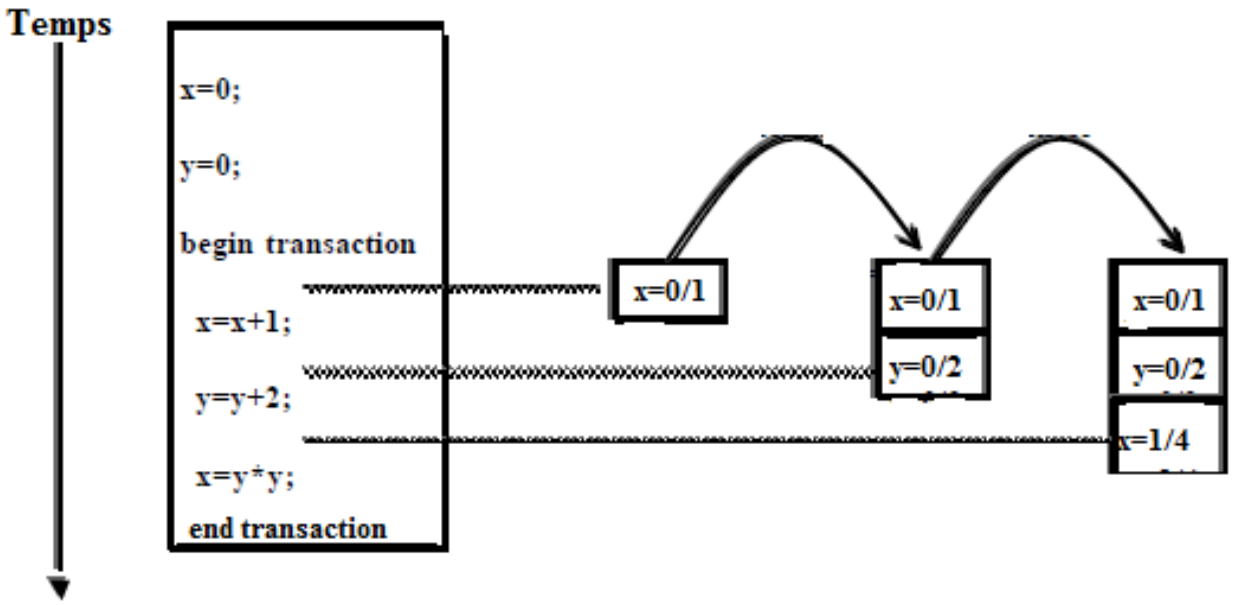
a) Mémoire d ’ombre (espace de travail privé)

Ne copier que les index de blocs et non les blocs des fichiers !

Contrôle de l'atomicité : « défaire » ou « refaire » en cas de défaillances

b)Utilisation de Journaux des inscriptions

Valeurs du journal « avant » (et stratégie de MAJ en continu) : avant de modifier une valeur, on l 'inscrit dans le log.



Contrôle de l'atomicité : « défaire » ou « refaire » en cas de défaillances

b)Utilisation de Journaux des inscriptions

Si la transaction valide, les modifications sur les fichiers ont eu lieu.
Si la transaction abandonne/plante, on peut se servir du journal avant pour défaire ce qui avait été fait avant le plantage (plantage pendant le calcul): ROLLBACK -- restauration

Avec un journal « après » et un espace de travail privé:
- les modifications n 'ont pas lieu sur les fichiers mais dans l 'espace de travail
- si plantage pendant la validation, le journal permet de finir ce que la transaction a promis de valider, cad, de finir de copier l 'espace de travail sur les vrais fichiers.

Validation atomique

- Action réalisée dans le cadre d'une transaction distribuée
- ✓ Accord entre tous les processus pour effectuer la requête de la transaction ou pas

- ✓ L'accord se passe de manière atomique pour éviter toute interférence avec d'autres transactions

- Avec les propriétés suivantes

- ✓ **Validité:** La décision prise est soit valider, soit annuler
- ✓ **Intégrité:** Un processus décide au plus une fois
- ✓ **Accord:** Tous les processus qui décident prendront au final la même décision
- ✓ **Terminaison:** Tout processus correct décide en un temps fini

Validation atomique: Principe général

- Un processus coordinateur envoie une demande de réalisation d'action à tous les processus
- Localement, chaque processus décide s'il sera capable ou non d'effectuer cette action
- Quand le coordinateur a reçu tous les décisions des processus, il diffuse la décision finale qui sera respectée par tous les processus

✓ Si tous les processus avaient répondu « oui », alors la décision est de valider et chaque processus exécutera l'action

✓ Si un processus au moins avait répondu « non », alors la décision est d'annuler et aucun processus n'exécutera l'action

➢ L'action doit être en effet exécutée par tous ou aucun

Validation atomique

- Validation atomique est simple a priori mais le devient bien moins dans un contexte de fautes …

- Contexte de fautes
  - ✓ Pannes franches de processus
  - ✓ Pertes de messages

- Pour que la validation fonctionne (plus ou moins bien) dans ce contexte, deux grands types d'algorithmes

  - ✓ Validation à 2 phases : 2 phases commit (2PC)
  - ✓ Validation à 3 phases : 3 phases commit (3PC)

Validation atomique : 2PC

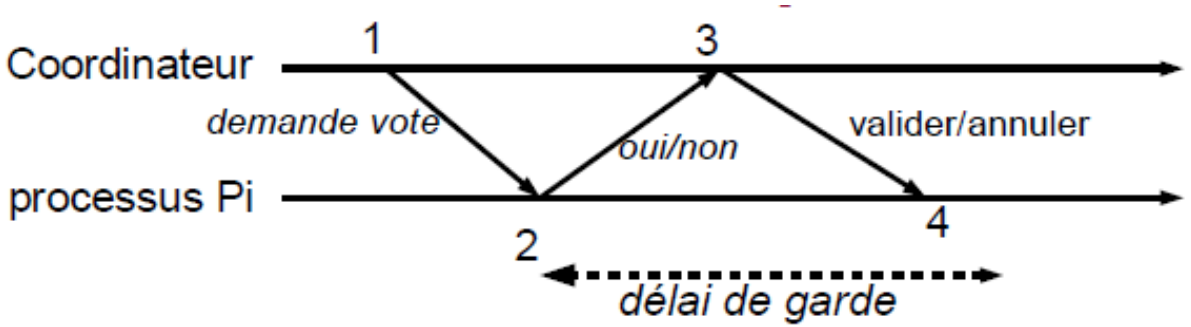
- Pour gérer perte de messages ou pannes franches
- ✓ Utilise un délai de garde : généralement un peu supérieur à 2 fois le temps de propagation d'un message

- ✓ En effet, bien souvent la communication prend la forme « envoi message » puis « réception acquittement »
  - ➢ Si pas reçu de réponse à l'envoi de message en 2 fois le temps de propagation : un message (ou l'acquittement) s'est perdu ou que le processus avec qui on communique est mort

- Validation à 2 phases (2PC)
- ✓ Première phase : demande de vote et réception des résultats
- ✓ Deuxième phase : diffusion de la décision à tous

Validation atomique : 2PC

1. Le processus coordinateur (un des processus parmi tous qui joue ce rôle) envoie une demande de vote à tous les processus
2. Chaque processus étudie la demande selon son contexte local et répond oui ou non (selon qu'il peut ou pas exécuter la requête demandée)
3. Quand le coordinateur a reçu tous les votes, il envoie la décision finale : valider ou annuler la requête
4. Chaque processus exécute la requête s'il reçoit valider ou ne fait rien s'il reçoit annuler



*Validation atomique 2PC:*Gestion des problèmes

- Si le coordinateur ne reçoit pas de réponse avant le délai de garde dela part de Pi
- ✓ Soit Pi est planté, soit la réponse de Pi est perdue ou la demande n'est pas arrivée à Pi
- ✓ Dans ce cas, on abandonne la transaction, la décision est annuler
  - ➢ On aurait pu relancer la demande à Pi, mais cela aurait retardé l'envoi de la décision finale aux autres processus et aurait déclencher d'autres problèmes

- Si la décision finale venant du coordinateur n'est pas reçue par Pi
- ✓ Le message à destination de Pi a été perdu
- ✓ Le coordinateur s'est planté et n'a pas pris ni envoyé de décision globale
- ✓ Pi peut interroger les autres processus pour savoir s'ils ont reçus une décision globale et laquelle
  - ➢ Si aucun processus ne peut lui répondre (soit ils ont rien reçu, soit ceux qui ont reçu se sont plantés)

Si Pi avait voté non : pas de problème, c'est forcément annuler

Si Pi avait voté oui : Pi n'a aucune idée de la décision finale qui a (peut-être) été prise

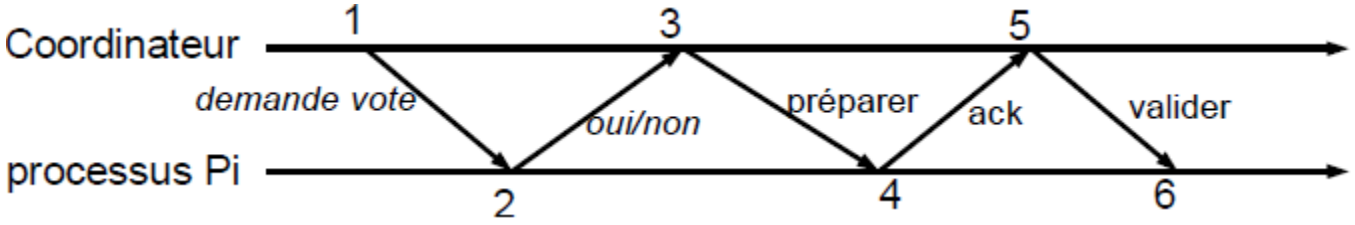
Validation atomique : vers la 3PC

- En conclusion sur la 2PC
- ✓ On peut se retrouver dans des situations où un processus ne sait pas quoi faire
- ✓ Le problème est qu'on attend pas de savoir que tout le monde est bien au courant de la décision finale avant d'exécuter ou pas la requête
  - ➢ La validation atomique à 3 phases rajoute une étape pour cela

- Validation atomique à trois phases (3PC)
- ✓ Première phase : demande de vote et attentes des votes (idemque pour 2PC)
- ✓ Deuxième phase : envoie à tous de la décision globale et attentede l'acquittement de la réception
- ✓ Troisième phase : diffusion de la demande d'exécution de la requête (sauf si décision annuler)

Validation atomique : 3PC

1. Le coordinateur envoie une demande de vote à tous les processus
2. Chaque processus étudie la demande selon son contexte local et répond oui ou non
3. Quand le coordinateur a reçu tous les votes, il envoie
- ✓ Soit annuler si un processus avait voté non
  - ➢ La validation est alors terminée, pas d'autres échanges de messages
- ✓ Soit une demande au processus de se préparer à exécuter la requête
4. Chaque processus envoie un acquittement au coordinateur de la réception de cette demande
5. Une fois tous les acquittements reçus, le coordinateur diffuse la validation finale
6. Chaque processus exécute alors la requête



Validation atomique 3PC: Gestion des problèmes

- Si un processus reçoit un « préparer » de la part du coordinateur, il sait alors que quoiqu'il arrive tous les processus ont décidé de valider la requête

- Donc même si l'acquittement qu'il a envoyé au coordinateur est perdu et/ou même si le processus ne reçoit pas le « valider » du coordinateur

✓ Le processus exécutera la requête car il sait que tout les processus avait décidé qu'ils pouvaient l'exécuter

Détection de terminaison d'une application répartie

Introduction

Ce n'est pas parce qu'un processus est momentanément passif (plus de messages à traiter en attente), que l'ensemble des processus de l'application est terminé
En effet , si un processus est encore actif, il peut tout à fait envoyer un message à un processus qui était passif

But

déterminer de manière complètement répartie un état global cohérent de l'application, ici, l' état de terminaison
Impossible de questionner les processus TOUS en MEME TEMPS !
Donc, difficile d 'obtenir une vision globale correcte en se basant uniquement sur des visions locales pas faites au même instant...

Détection de la terminaison

Principe

Méthodes permettant de passer d'un état message terminaison à un état process terminaison. Basées sur le schéma suivant :
1 détection de la configuration terminale
2 diffusion de l'information à tous les processus
3 passage dans un état terminal.

Phases

Algorithme de base : gestion de l'état des processus,
Algorithmes de contrôle : ajoutés pour gérer la terminaison,

Algorithme de base

Etat des processus

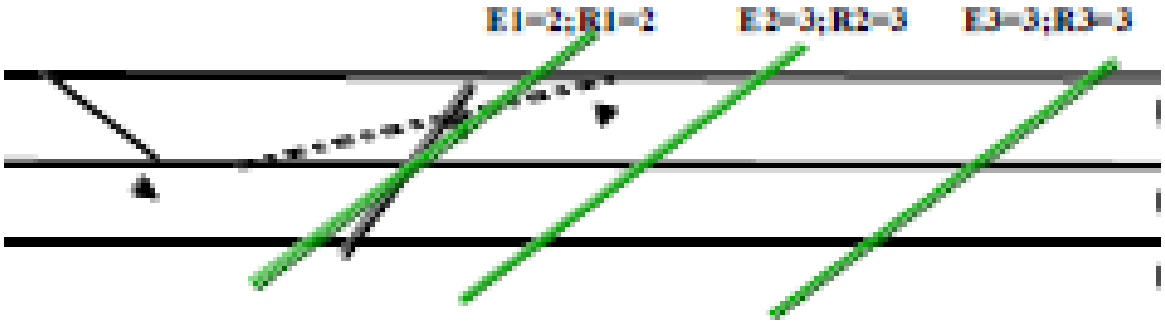
- Deux états possibles : etatp = (actif ; passif)
- Un processus est actif (resp. passif) si un événement interne ou une réception de message est (resp. n'est pas) accessible à son état.
- Un processus p passif n'admet que des réceptions et peut à tout instant devenir actif à la reception d'un message.

Règles de changement d'état des processus

Un processus actif devient passif uniquement suite à un événement interne.
Un processus devient actif à la suite de la réception d'un message.
Les événements suite auxquels p devient passif sont des événements internes à p.

Algorithme des 4 compteurs (Mattern)

•Comptabiliser, sur chaque processus, le nombre de message envoyés et reçus
•Sommer ces nombres
•Si E=R, alors, ce n'est pas forcé que ce soit fini
•Donc, recommencer une seconde collecte des compteurs locaux
•Si E1=R1=E2=R2 (4 valeurs), Alors , l'application est bien terminé



•La vague de collecte peut doubler des messages de l'application (si par ex, voies entre 2 processus non FIFO)
•Quand c'est fini, il faut 2 vagues pour le constater

Algorithme de visite des sites (Actif/Passif), sur un anneau (Misra 1983)

Hypothèses et principe

• aucune hypothèse sur la topologie des voies de communication ni sur le délai de transfert des messages.

• Les seules hypothèses sont :
- pas de perte des messages
-pas de déséquencement (canaux FIFO)

• La terminaison est réalisée lorsque tous les processus sont passifs et qu'il n'y a plus de messages en transit.

• L'algorithme de Misra consiste aussi à faire visiter les processus par un jeton. la constatation par le jeton que tous les processus sont passifs lorsqu'ils ont été visités ne permet pas de conclure à la terminaison : des processus ont pu être réactivés et des messages peuvent être en transit.

Algorithme de visite des sites (Actif/Passif), sur un anneau (Misra 1983) (suite)

• Dans le cas particulier où la topologie de communication entre processus est un anneau, le jeton peut affirmer que le calcul est terminé si, après avoir effectué un tour sur l'anneau, il constate que chaque processus est resté passif depuis sa dernière visite à ce processus.

• En effet, les messages ne pouvant pas se doubler, entre deux visites du jeton un processus a nécessairement reçu les messages qui étaient en transit lors de la première visite du jeton.

⇒ si le jeton a effectué deux tours sur l'anneau et qu'il n'a observé que des processus restés en permanence passifs, on peut conclure à la passivité des processus et à l'absence de messages en transit, i.e à la terminaison.

Hypothèses et principes

Comportement du jeton

• lorsqu'un processus devient actif il devient noir, et c'est le jeton qui le peint en blanc lorsqu'il quitte le processus (passif).

• ainsi si le jeton retrouve blanc un processus, c'est que ce dernier est resté passif en permanence depuis sa dernière visite.

• lorsque le jeton a visité les n processus et qu'il les a trouvés tous blancs, il peut en conclure la terminaison (initialement les processus sont noirs).

• tous les processus ont des comportements identiques

Hypothèses et principes (suite)

Pour admettre une topologie quelconque pour les communications et ne pas se limiter à un anneau de contrôle, le jeton doit :
- visiter tous les processus,
-s'assurer qu'il n'y a plus de messages en transit.

⇒ le jeton doit parcourir chaque arc du réseau

⇒ si le réseau est fortement connexe alors il existe un circuit C qui comporte chaque arc du réseau (au moins une fois).

⇒ il suffit alors de remplacer l'anneau par un tel circuit

L'algorithme

Le jeton véhicule une valeur j : les processus visités lors des j dernières traversées des voies de communication sont restés en permanence passifs.
On suppose que sur le circuit C (pré-calculé) les fonctions suivantes sont définies :
- fonction *taille (C : circuit) résultat entier : donne la taille du circuit*
- fonction *successeur (C : circuit, i : 1 .. n) résultat 1 .. n : donne pour le* processus Pi qui l'exécute, le numéro du successeur de Pi sur le circuit.
Le jeton détectera la terminaison lorsque l'on aura : *j = taille (C)*
Chaque processus Pi est doté des déclarations suivantes :
var couleur : (blanc, noir) initialisé à noir ;
état : (actif, passif) initialisé à actif ;
jeton_présent : booléen initialisé à faux ;
nb : entier initialisé à 0 ;
- la variable « *couleur » est associée au processus et « nb » sert à mémoriser la* valeur associée au jeton entre sa réception et sa réémission.
- les messages sont de deux types : *messages, jeton*.

L’algorithme

Début
⇒ lors de réception de (message, m) faire
début
état → actif ;
couleur → noir ;
fin
⇒ lors de attente (message, m) faire
début
état → passif;
fin

L’algorithme

⇒ lors de réception de (jeton, j) faire
début
nb →j ;
jeton_présent →vrai;
si nb = taille (C) et couleur = blanc alors
terminaison détectée
fsi ;
fin

L’algorithme

⇒ lors de émission de (jeton, j) faire
début
si jeton_présent et état = passif alors
début
si couleur = noir alors nb → 0
sinon nb → nb + 1 ;
fsi ;
envoyer (jeton, nb) à successeur (C, i) ;
couleur → blanc ;
jeton_présent → faux ;
fin
fsi
fin
Fin

Conclusion

• l'algo de Misra peut être généralisé au cas où le réseau de communication n'est pas fortement connexe : le réseau est décomposé en ses composantes maximales fortement connexes, et le jeton visite alors successivement ces composantes selon un ordre prédéfini.

• il existe d'autres algorithmes qui utilisent l'estampillage (algo de Rana) : le principe de cet algo est voisin du principe d'extinction sélective des messages de l'algo d'élection de Chang et Roberts :
⇒ il s'agit d'élire le dernier processus terminé qui diffusera ensuite l'information *terminé*.

Merci pour votre attention